

Jigsaw

An infrastructure for cross-device mashups

Master Thesis

Stefano Pongelli

<stefano@student.ethz.ch>

Prof. Dr. Moira C. Norrie
Maria Husmann, Dr. Michael Nebeling

Global Information Systems Group
Institute of Information Systems
Department of Computer Science

6th November 2013



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Abstract

The ubiquity of web applications and the increasing proliferation of computing devices in everyday life lead to situations and environments where collaboration and reuse of existing web-based resources become ever more important. While research usually focuses either on composing parts of different web sites to provide new applications (mashups) or distributing user interfaces across devices and users (DUIs), little work has been done in bringing the two research streams together. The existing solutions, furthermore, are complex and targeted at experienced developers.

Sitting at the intersection of the two research areas of mashups and DUIs, as well as integrating aspects of collaborative browsing, this thesis presents Jigsaw—an infrastructure for cross-device mashups. The focus in Jigsaw is on direct manipulation techniques for existing web applications designed to enable developers as well as end-users to create DUIs by distributing and combining multiple web sites across multiple devices. Jigsaw is built upon a client-server remote control architecture that, by using a headless browser, provides synchronisation as well as continuity between devices, mashups, and web pages that are distributed as components.

In a two-step evaluation involving both technical and conceptual considerations, we investigate Jigsaw’s breadth by testing a simple but representative use case scenario on a variety of top web sites selected according to popularity and traffic, and we discuss a number of dimensions defined by a logical framework for multi-device environments extended to cross-device mashups. Our results suggest that most web sites would work without major issues and could, therefore, be used in cross-device mashups based on Jigsaw.

Acknowledgements

First of all, I would like to express my appreciation to Prof. Moira Norrie and the GlobIS group as a whole, not only for giving me the opportunity to write this thesis, but also for the very interesting and challenging courses and semester projects offered at ETH Zürich. Moreover, I would like to thank in particular my supervisors Maria Husmann and Michael Nebeling for their support and guidance during the course of this thesis.

Finally, I would like to thank my friends and family for bearing with my occasional ranting, and especially my father, for introducing me to the world of computers many years ago, when starting a game required typing strange and magical sequences of DOS commands into a scary black screen.

Contents

1	Introduction	1
2	Background	5
2.1	Mashups	5
2.2	Distributed User Interfaces	7
2.3	Collaborative Browsing	8
2.4	Remote Control Architecture	9
2.5	MultiMasher	10
3	Cross-Device Mashups	11
3.1	Requirements	12
3.1.1	Entities	12
3.1.2	Operations	14
3.2	Scenarios	18
3.2.1	Shared Playground	18
3.2.2	Living Room News	20
4	Jigsaw	21
4.1	Home Page	23
4.2	Mashup	24
4.2.1	Creating Components	24
4.2.2	Component Modes	25
4.2.3	Component Configuration	25
4.2.4	Co-browsing Feedback	28
4.3	Overview	29
4.3.1	Interactive Previews	29
4.3.2	List of Components	30
4.3.3	List of Clients	30
5	Architecture	31
5.1	State Consistency	31
5.2	Remote Control	32
5.3	Server Logic	33
5.3.1	Headless Browser	33
5.3.2	Clients	34
5.4	Data Model	36
5.5	Thin Client	36

6	Implementation	39
6.1	Technologies	39
6.1.1	Server-side	40
6.1.2	Client-side	42
6.2	Communication	43
6.2.1	Protocols	43
6.2.2	Events	45
6.3	Server	48
6.3.1	Mashup Logic	48
6.3.2	Component Logic	50
6.3.3	Page Logic	50
6.3.4	Browser Logic	52
6.3.5	Source Extraction	54
6.4	Client	57
6.4.1	Controllers	58
6.4.2	Directives	61
6.4.3	HTML	65
7	Technical Evaluation	67
7.1	Criteria	67
7.2	Scenario	69
7.3	Results	70
8	Conceptual Evaluation	75
8.1	Framework	75
8.1.1	Multi-Device Environments	75
8.1.2	Cross-Device Mashups	76
8.2	Discussion	77
9	Conclusions	81
	Appendices	89
A	User Manual	89
A.1	Requirements	89
A.2	Installation	89
A.3	Configuration	90
B	Evaluation Data	91

1

Introduction

The Indexed Web contains at least 3.8 billion pages¹ which are accessible from anywhere at any time, through a number of different devices ranging from desktop computers to mobile phones. The research area of mashups explores design methods and models for taking advantage of existing web applications, by integrating content and functionality from different web sources [4]. However, this integration often requires a specific toolkit, and the resulting applications are usually designed for single-user/single-device scenarios [5].

The increasing proliferation of computing devices in everyday life leads to environments in which it is not uncommon for a person to own a mobile phone, tablet, desktop, and laptop computer [8, 15]. At the same time, there are devices that are shared by a group of users such as interactive tabletops or large screens. These two factors combined produce ecosystems that contain multiple users and multiple devices, some shared and some private. Furthermore, devices are often interconnected through a network and to the internet; it is therefore desirable to use them in combination, rather than in isolation. Research in the area of distributed user interfaces (DUIs) addresses this issue providing frameworks and tools to systematically support the distribution and migration of applications across devices [9, 19].

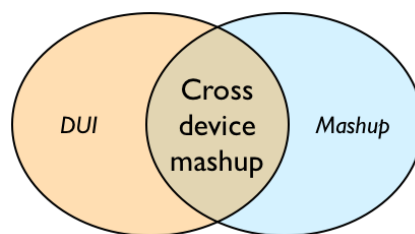


Figure 1.1: Combination of DUI and mashup: cross-device mashup

¹<http://www.worldwidewebsize.com>

Bringing together the two research areas of DUIs and mashups, as shown in Figure 1.1, this thesis focuses on building an infrastructure that enables developers to create DUIs by distributing and combining multiple web sites across multiple devices. We will henceforth call these kind of web applications *cross-device mashups*. The concept of cross-device mashups rises a number of interesting challenges: on one hand, technical solutions in the field of DUIs are usually limited to considering only a single application; on the other hand research on mashups has so far focused on combining multiple existing web applications, but on a single device. An infrastructure that unifies the two research areas would not only need to take into account existing problems and solutions of the respective fields, but also new problems arising from the combination of the two.

In related work, the development of a new paradigm defined as distributed UI orchestration targets similar challenges, allowing integrating web services and UI across multiple devices [6]. However, the intrinsic complexity of UI orchestrations prevents less skilled developers or end users to take advantage of such platforms. In order to enable developers to rapidly create DUIs, without having to spend a significant amount of time training, *MultiMasher* [14] provides the foundation for this thesis by conceptualizing a visual tool for designing cross-device mashups, as well as addressing some of the arising challenges.

In this thesis, we present *Jigsaw*, an infrastructure for cross-device mashups which, focusing on direct manipulation of existing web applications, enables developers and end users to create DUIs by distributing and combining multiple web sites across multiple devices. This thesis, in particular, realizes and expands upon cross-device mashup concepts by providing an infrastructure which aims at solving technical problems like state synchronisation and interactions between different devices and web sites, as well as problems emerging from the collaborative environment. Mashups typically assume a single user, which does not necessarily hold for cross-device mashups. *Jigsaw*, therefore, not only needs to be aware of the available devices and their characteristics, but it also has to rise awareness for cross-device interactions originating from different users. This is a new requirement: since the user interface is spread across multiple devices, an interaction on one device may trigger an update to the interface on another device.

The main contributions of this thesis are the following:

- Advancing the concepts introduced in *MultiMasher*, we propose a solution that combining the research areas of DUIs and mashups focuses on enabling developers to rapidly create DUIs by distributing and combining multiple web sites across multiple devices. This solution addresses technical challenges such as state synchronisation and interactions between different devices and web sites, as well as challenges emerging from the multi-user, collaborative environment.
- We present the design and implementation of *Jigsaw*—an infrastructure for cross-device mashups which, addressing the challenges arising from bringing together the two research areas, realizes the aforementioned solution.
- We report on a technical evaluation assessing the behaviour of 50 most visited web sites loaded in our infrastructure, as well as the effectiveness of the distribution and mashup capabilities.

- We discuss and expand upon design dimensions identified in an existing framework, which can help in better understanding the features provided by tools and applications for multi-device environments in general, and cross-device mashups in particular.

The rest of this thesis is organized as follows. Chapter 2 discusses the background of this project and reviews related work. Chapter 3 elaborates on the approach towards cross-device mashups by providing requirements for such environments, as well as possible use case scenarios. Chapter 4 describes the main features offered by Jigsaw. Chapter 5 introduces the concepts and architecture supporting the proposed infrastructure. Chapter 6 provides an overview of the technologies used, followed by implementation details. Chapter 7 reports the results of a technical evaluation in which we tested our infrastructure with top 50 web sites selected according to popularity and traffic, while Chapter 8 discusses and evaluates design dimensions for cross-device mashups at a conceptual level. Finally, Chapter 9 exhibits conclusions as well as future work.

2

Background

Bringing together the two research areas of UIs and mashups, this thesis focuses on building an infrastructure that enables developers to rapidly create UIs by distributing and combining multiple web sites across multiple devices. Multi-device environments are particularly suited for collaborative tasks: this often implies complex interactions between the various devices, as well as between the running applications.

This chapter provides background information as well as examples of state of the art developments in the research areas of mashups, UIs, and collaborative environments. Furthermore, it shows some examples of remote control architectures, which are central to the developed infrastructure. Finally, it introduces the basis and inspiration for this project: *MultiMasher*, a visual tool for multi-device mashups.

2.1 Mashups

A mashup is the result of techniques for building applications, which combine content from multiple sources to create an integrated experience. These kind of web applications have gained popularity since the rise of Web 2.0. In fact, one of the first examples of mashups has been created in 2005: *HousingMaps*¹ combines data coming from Craigslist² with the public Google Maps API³, showing for sale properties on a map. Nowadays, there is a great body of research focusing on mashups; however, the solutions are usually designed for single-user/single-device scenarios.

In Ghiani et al. [10] the authors present an environment to enable people without programming knowledge to create mashups composed of web components, broadly defined as parts

¹<http://housingmaps.com>

²<http://www.craigslist.org>

³<https://developers.google.com/maps>

of a web page, that can be selected directly from existing web applications by specifying the ID of the topmost element in the desired DOM tree. Their environment requires the use of a proxy server for accessing the web applications, which are then injected with scripts to enable the selection of the components that will be part of the newly created mashup. The inter-component communication is then managed by intercepting HTTP requests, in particular by identifying all the parameters associated with an output component. It is then possible for users to select any of these parameters and indicate that the value entered in a given field in an input component should feed the selected input parameter. In *Jigsaw*, we follow the same principle of allowing users to create mashups through direct manipulation of existing web applications, without requiring the use of a specific browser and the knowledge of any programming language. Instead of selecting components by the ID attribute (and assigning custom IDs when missing), we use an algorithm for computing the CSS path to the selected DOM element, thus avoiding the explicit need for IDs and allowing complete freedom of selection. Both approaches, however, cannot address issues coming from evolving web sites, which might change elements IDs or even their DOM structure.

A more sophisticated approach, targeted at creating web components by extracting DOM from existing web pages, is described in Firecrow [18]. Firecrow aims to extract and reuse UI controls with a semi-automatic method: the developer selects the control and performs a series of interactions that represent the desired behaviour; in the background, the execution is analysed and all code (HTML, CSS, JavaScript) and resources necessary for the standalone functioning of the control are extracted. This method is limited to the extraction of basic standard controls of web applications, without the goal of obtaining dynamic mashups.

Research in user interface mashups has been advanced in Chudnovskyy et al. [3] by focusing on inter-widget communication (IWC). These widget, defined as small, self-contained, single-purpose web applications, are built by different vendors and rely on different data models. Therefore, IWC rarely works "out of the box", which leaves users with the tedious task of manual wiring and limited functionality. To address this problem, the authors developed a semi-automatic approach which, employing the programming by demonstration paradigm, extends widgets with IWC capabilities at the presentation layer. While a semi-automatic approach could be useful, it could also lead to confusion and unwanted connections between components when used in multi-user environments. Taking inspiration from this presentation layer IWC system and the query parameters selection used in Ghiani et al. [10], our infrastructure allows inter-component communication by letting users augment HTML elements with an action (click, change, submit) and a tag. Elements with the same tag and action will then be synchronised, independently from the component they belong to.

Finally, in mashArt [5] the authors propose universal composition as a service, in form of an easy-to-use graphical development tool equipped with an execution environment for fast deployment and execution of composite web applications. The presented component model, based on the premise that components should be created by experts and treated as widgets, notes that synchronization should be adopted as interaction paradigm using four abstractions: state, events, operations, and properties. A component should therefore have a state, be able to

handle user events as well as react to the responses (operations), and have properties defining setup information like position and style.

2.2 Distributed User Interfaces

Distributed user interfaces (DUIs) have recently become a new field of research and development in Human-Computer Interaction (HCI). DUIs bring about drastic changes affecting the way interactive systems are conceived: they go beyond the fact that user interfaces are controlled by a single end user, on the same computing platform, in the same environment. This new interaction mechanism affects the way these novel systems are designed and developed. New features need to be taken into account from the very beginning of the development process and in general new models, methods, and tools need to be considered for the development of interactive systems based on distributed user interfaces.

Focusing on the migration of web applications, Ghiani et al. [11] presents an environment which supports dynamic push and pull of interactive web applications, or parts of them, across desktop and mobile devices while preserving their state. Users can therefore start working on a part of a web site, push it to a different device and continue working on that device. Furthermore, users can request a web site which is used by a different device with a pull request. The system uses a proxy based approach: pages are annotated (by injecting JavaScript) in the proxy before being sent to the clients. In order to maintain state synchronisation, the injected JavaScript separates the original DOM from the modified content (e.g. compiled forms). While this platform is interesting for migration purposes, it does not take into account collaborative browsing and does not support interaction between different web sites.

In *DireWolf* [16] the authors explored the concept of distribution and migration with a framework targeted at web applications based on pre-built widgets. These widgets are usually designed to accomplish small stand-alone tasks, which may recur in multiple different applications: with this framework they can be augmented to exchange information. However, *DireWolf* aims at single-user/multi-device scenarios, hence, similarly to Ghiani et al. [11], it does not take into account collaborative browsing.

A more elaborate paradigm defined as *distributed UI orchestration* [6] targets challenges similar to those expressed in this thesis, allowing integrating web services and UI across multiple devices. However, the intrinsic complexity of UI orchestrations prevents less skilled developers or end users to take advantage of such platforms. A number of concepts and requirements expressed in that paper are adopted in this thesis: firstly, we provide distributed UI synchronisation without the modelling complexity introduced by BPEL⁴; secondly, we adapt the concept of pages hosting place holders for UI components into the concept of mashups hosting freely positionable UI components. Finally, we explore the concept of UI component itself, and the fact that events of a UI component may be propagated to components running in the same web page or in other pages of the application. However, while *distributed UI orchestration* requires skilled developers to cover all development

⁴<http://searchsoa.techtarget.com/definition/BPEL>

aspects in an integrated fashion (i.e. design, deployment, and execution support), this thesis focuses on performing the aforementioned tasks seamlessly, without having to program or model complex orchestration.

Particularly related to the conceptual evaluation provided in this thesis, Paternò and Santoro [22] presents a framework for describing various design dimensions that can help in better understanding the features provided by tools and applications for multi-device environments. These dimensions range from UI distribution (static or dynamic) and UI migration (continuity) to user interaction modalities and architecture (client/server, p2p). We will use the dimensions presented in the paper in order to evaluate our infrastructure. We will however extend and adapt them to better represent our collaborative, multi-device/multi-user environment based on mashups.

2.3 Collaborative Browsing

Collaborative browsing, or co-browsing, is the co-navigation of the Web with other people, supported by software that takes care of synchronizing the browsers. The majority of web applications only facilitates single-user support. Adding shared capabilities to existing single-user applications may often require complex concurrency control systems which are costly to implement, test and maintain. Co-browsing is a central aspect of our infrastructure: once web sites are distributed as components and mashed together, multiple users may interact with the same component in collaboration. Research in the area of co-browsing is not new [12], however, challenges are still present today, especially when synchronization of JavaScript interactions is taken into account.

The issue of providing co-browsing of dynamic, JavaScripted web pages is tackled in Lowet and Goergen [17]. The presented solution works at the DOM level, can be implemented in JavaScript and requires no extensions to the browser. In particular, the paper describes two mechanisms: JavaScript engine input synchronization and JavaScript engine output synchronization. The first one takes into consideration UI events alone, thus synchronisation happens before the execution of JavaScript. The latter synchronises the changes made on the DOM tree, after the JavaScript execution. In addition, both mechanisms use a reference browser to synchronise the web page state. Interestingly, the authors of the paper after evaluating robustness, user experience, scalability and implementation effort decided to use JavaScript engine input synchronization. However, in this thesis we present a system which uses similar concepts as those expressed in JavaScript engine output synchronization: while the other approach is better for scalability and user experience, it is not clear whether it takes into account the fact that propagation of interactions may generate multiple updates to a database, for instance when submitting a form.

In Heinrich et al. [13], the authors focus on web editors by describing a generic transformation approach for web applications capable of turning single-user editors into multi-user editors supporting shared editing. In particular, they propose a generic collaboration adapter and an operational transformation engine. While the generic collaboration adapter is capable of tracking DOM manipulations (e.g. create, delete, and modify DOM elements), the operational transformation engine provides a synchronization service as well as a

conflict resolution mechanism based on a concurrency control algorithm called operational transformation. In our infrastructure, since it aims at being compatible with a vast variety of web sites, we decided to not provide concurrency control mechanisms. However, such algorithms could be added in the future.

Finally, *PlayByPlay* [25] is a general purpose web collaboration tool that, augmenting a web site, uses the communication model of instant messaging to support a variety of collaborative browsing tasks. As the users interact with their web browsers, their interactions are recorded as natural language descriptions and sent to the other users via the chat channel. Moreover, users can exchange clip and snippets of web content. While this approach is useful for remote collaboration, our project aims at co-located user interaction, where users can talk with each other.

2.4 Remote Control Architecture

The project presented in this thesis uses a remote control metaphor to provide seamless state synchronisation across devices, as well as avoiding the aforementioned problem of multiple updates being erroneously generated. Remote control architectures are common, the most famous being *VNC*⁵, which allows remote desktop access. Other systems, like *WinCuts* [23], allow portions of an application to be remotely controlled, enabling users to re-author an existing application by choosing and re-arranging its on-screen rendering. However, those tools use screen shots to show the state of the remote system, while they replicate the exact mouse and keyboard events from the controlling system on the controlled system. We are interested in the realm of browsers: using images would mean loosing the existing HTML and CSS features (e.g. hovering a link).

In related work, *Highlight* [21] provides a server-side architecture that enables rapid prototyping and deployment of mobile web applications created from existing web sites. To do so, they use a remote control metaphor in which the mobile device controls a fully functional browser that is embedded within a proxy server. Content is then clipped from the proxy browser, transformed if necessary, and then sent to the mobile device as a typical web page. Events from the mobile device are then translated into interactions with the proxy browser. The presented system, however, only deals with a single web site and still requires a fair amount of programming. Furthermore, it uses an actual browser (Mozilla Firefox) within the proxy server: this makes handling page events and interactions quite laborious.

The description of a virtual browser architecture is provided in Cheng [2]. The main idea behind the proposed virtual browser is that, acting as a proxy, it could separate logical workflow and content display of a web application. Furthermore, using its own JavaScript engine to execute the logical workflow, it could generate a single DOM tree in a centralized way, and then dynamically split it into multiple sub DOM trees and distribute them to the corresponding devices for displaying. Creating such a browser however would be quite complex and time-consuming. Since we do not need a similarly convoluted browser, in our project we decided for a midway between this approach and the aforementioned use of a real browser.

⁵www.realvnc.com

2.5 MultiMasher

MultiMasher [14] provides foundation for this thesis both in terms of idea and concepts, by proposing a tool to enable multiple devices to access multiple web sources in a coordinated manner, thus creating multi-device mashups. Of particular interest is the concept of facilitating the quick and easy creation of multi-device mashups through a direct manipulation interface, without the need for modelling or programming. The principal feature of *MultiMasher* is augmenting a web site with a distribution toolbar, by injecting JavaScript into the page. From the toolbar, users can see the available devices, select elements to distribute by entering a *design mode* and then use the distributed elements collaboratively in *distribution mode*. Similarly to Ghiani et al. [10], elements are selected by ID, however the selection is done by highlighting the specified element, without the need for programming. In order to manage collaborative interactions (co-browsing) between devices, a mechanism similar to the JavaScript engine input synchronization [17] is used. Therefore, DOM events originating in a device are replayed into the other devices, thus providing a partial solution to synchronising state.

In this thesis, we present *Jigsaw*, an infrastructure which fulfils the goals of *MultiMasher* while extending them to solve some of the challenges introduced by creating and distributing mashups in multi-device, multi-user, collaborative environments. The next chapter introduces the concept of *cross-device mashups* which is central to the presented infrastructure, as well as providing some scenarios.

3

Cross-Device Mashups

In order to characterize the environment obtained by bringing together the research areas of mashups, DUIs and collaborative browsing, we advance the concept of cross-device mashup. Broadly defined, a cross-device mashup is a collection of mashups that are distributed among multiple cooperating devices. These devices may therefore access multiple web sources in a coordinated manner.

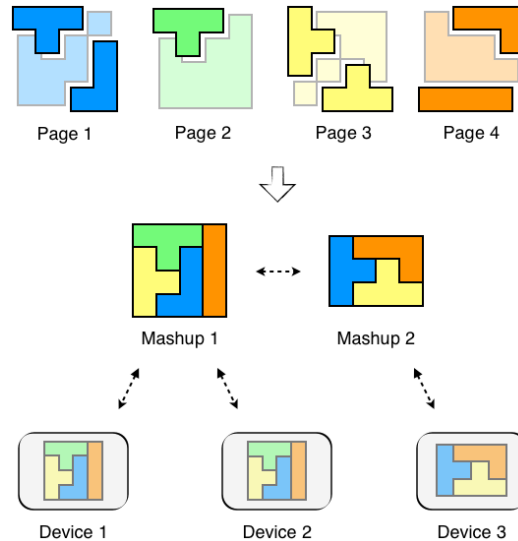


Figure 3.1: Overview of a cross-device mashup

In the context of this thesis, in particular, a mashup is described as the combination of one or more components, each encapsulating a web page or parts of a web page. Figure 3.1 shows pages split into components, which are then used to build mashups that are finally used by the devices.

In the following section we will highlight a number of general requirements, most of which are supported by Jigsaw, that a cross-device mashup should satisfy. Moreover, we will illustrate some possible use case scenarios.

3.1 Requirements

Conceptually, a cross-device mashup can be better characterized by defining the main entities required, as well as a number of operations that are typical of distributed user interfaces, mashups, and collaborative browsing.

3.1.1 Entities

We define four principal entities that characterize this environment: distributed components, exposing web pages; mashups, combining the components; devices, interacting with the mashups; and users, handling the devices.

Components

Cross-device mashups combine existing web pages provided by external sources. Components, which are the fundamental building blocks of mashups, should encapsulate these web pages. However, users might not necessarily want to use the entire document, but only certain DOM elements. To address this issue, we define the concept of *view* exemplified in Figure 3.2.

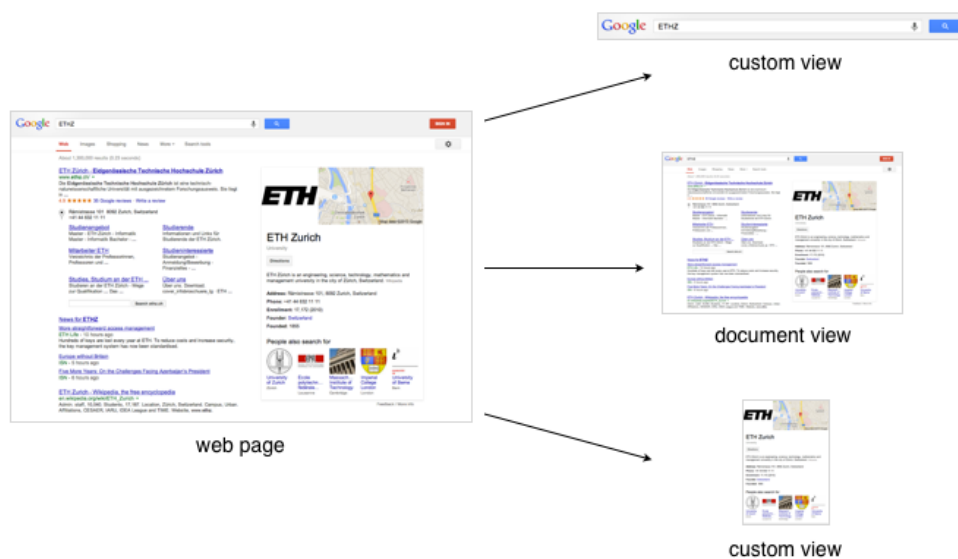


Figure 3.2: Web page divided into views

A web page may be split into a number of custom views, which only show particular DOM elements: for instance, a view might show the menu, while another one might show the content of the page. Following this logic, an entire web page is represented by a document view.

A *component*, therefore, is defined as the container of a web page with all its views. In particular, it should be an interactive element which, having specific properties, might be freely placed, resized and moved in the context of a mashup, as well as across mashups. Furthermore, components must be used by devices to interact with views; they should therefore show a single view at the time (e.g. the search bar). On the other hand, multiple components might use the same page while showing different views (Figure 3.3), and communicate with each other. Finally, in the scope of this thesis, components should handle web pages at the UI layer.

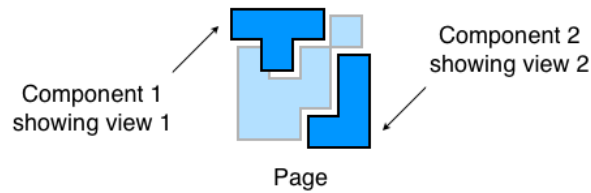


Figure 3.3: Two components showing different views of the same web page

Mashups

A *mashup* is a web page resulting from the combination of one or more components which might show specific views of a number of web pages, as previously illustrated in Figure 3.1. Mashups might be accessed and used by a number of devices simultaneously, in collaboration. Moreover, they should be able to communicate with each other through linked components, for instance in order to automatically replay a user interaction, like a search query, in multiple web pages. Figure 3.4 shows two communicating mashups that are using components which expose views of different pages.

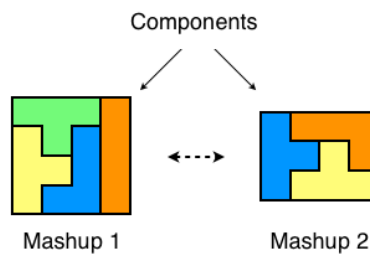


Figure 3.4: Two mashups with components from different pages

Devices

In principle, any kind of *device* that can use a browser, for instance Google Chrome or Safari, should be able to create, access, and manage the cross-device mashup without having to install any additional software. A device might therefore be a desktop or laptop computer, a tablet, a smartphone, or even a Smart TV connected to the local area network. As shown in Figure 3.5, devices might not communicate directly together, but through the cross-device

mashup by interacting with components. However, solutions supporting peer-to-peer communication might be taken into account for providing further functionalities. For instance, in the collaborative spectrum, it might make sense to use a separate communication channel for instant messaging between devices.

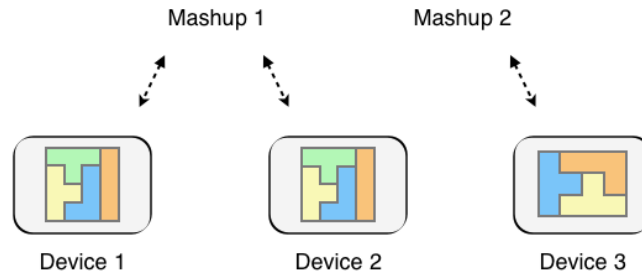


Figure 3.5: Three devices using to a cross-device mashup

Users

Cross-device mashups are multi user environments. The *users* should therefore be able to collaboratively browse the mashups, and cooperatively interact with their components.

Furthermore, users might be explicitly or implicitly defined. In the former case, the cross-device mashup should support proper login mechanisms, as well as take into account privacy issues. In the latter case the cross-device mashup might implicitly define anonymous users by considering each device, or even each device session, as a different one: privacy, therefore, might be less relevant. In both cases, however, the cross-device mashup should account for security issues. Furthermore, it might provide access control, either based on devices, users or both.

3.1.2 Operations

After having defined the main conceptual entities, we propose a number of operations which a cross-device mashup should support. These operations are typical of mashups at the UI layer, distributed user interfaces, and collaborative browsing. We will therefore categorise them under *mashing*, *distributing*, and *co-browsing*.

Mashing

In general, multiple devices should be able to create, load, use, and freely switch between mashups by selecting the desired one.

In the area of mashups, furthermore, developers usually deal with inter-communicating components which are either self contained widgets [3], or pieces of DOM extracted from web pages [18]. Creating a cross-device mashup, in particular, implies applying the concepts of mashups to those of distributed user interfaces, i.e. the distributed elements should be components that encapsulate views of different web sites, as defined in Section 3.1.1.

Therefore, we define the following mashing operations:

Component Creation It should be possible to create components, either programmatically [21] or by direct manipulation of web sites [10], and assign them to mashups. Components using the same underlying web page should maintain state synchronisation, i.e. when the DOM of the web page is modified by either a user interaction or in any other way (e.g. AJAX call), both components should be consistently updated.

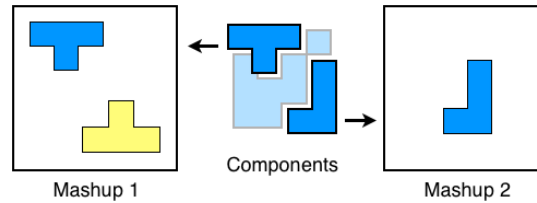


Figure 3.6: A web page divided into components assigned to two mashups

Inter-Component Communication Components should be able to communicate between each other: this communication might be implicit in case the components are using the same underlying web page, or explicit in case they are using different web pages. Explicit inter-component communication may be based on user interactions (e.g. clicks), or on DOM events in general. In the latter case, components should be able to observe each other and automatically trigger events (e.g. “if DOM element X of component Y changes, do something”).

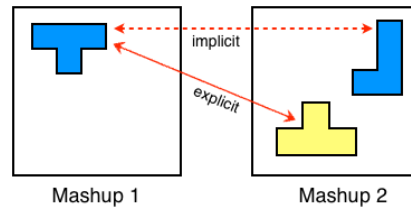


Figure 3.7: Inter-Component communication, implicit and explicit

Distributing

As previously mentioned, both mashups and their components are distributed: the former among devices, while the latter among mashups. In this context, the concept of *distribution* is therefore of great importance.

Components, in particular, might be added, copied, cloned, and deleted. Moreover, it should be possible to migrate components across mashups by using push and pull mechanisms similar to those presented in Ghiani et al. [11]. Finally, basic adaptivity operations dealing with components might include move and resize, while more advanced adaptivity features might even target the underlying web pages, in similar ways to what suggested in Nebeling et al. [20].

A cross-device mashup, therefore, should support the following operations with components:

Clone In order to create multiple components that share the same web page, it should be possible to clone them. The resulting component should maintain the views of the original one and use the same instance of the underlying web page. Therefore, DOM changes in the page would be synchronised across the cloned components. Cloning components would not only be useful for rapidly showing different views of the same web page, but also for migrating components to other mashups while keeping the original intact.

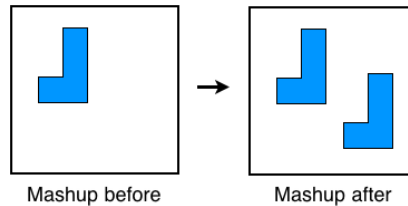


Figure 3.8: New component resulting from cloning an existing one

Copy Expanding on the concept of reuse, it should be possible to obtain a new instance of an existing web page by copying an existing component. The difference between creating a new component from scratch, is that the copied component would maintain the previously created views. However, DOM changes in the underlying web page would not be synchronised across the copied components.

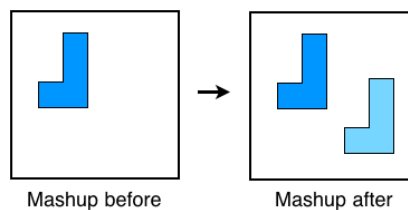


Figure 3.9: New component resulting from copying an existing one

Delete It should be possible to delete (i.e. destroy) components from the mashups, for instance those that are not needed any more. The underlying web page should not be lost, and it may therefore still be used by other components. The particular view used by the removed component, as well as all its other existing views, should remain available to other components using that page.

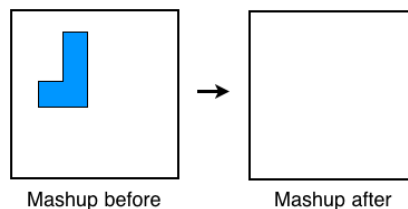


Figure 3.10: Removing components from mashups

Move As part of the possible adaptivity features, instead of being placed into fixed place holders components should be movable inside a mashup. We believe that in combination with the ability of resizing components, this would offer the possibility for a vast variety of compositions.

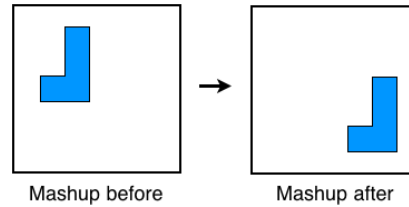


Figure 3.11: Moving components

Resize Complementing the aforementioned move operation, components should be resizable. Multiple resizing modalities should be considered. For instance it might be possible to change the size of the component by adding scroll bars, or by scaling the web page itself.

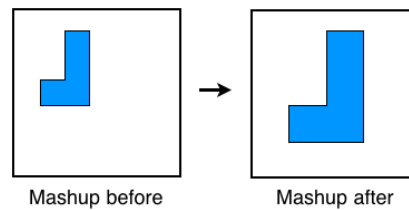


Figure 3.12: Resizing components

Migrate Finally, it should be possible to move components from one mashup to another (component migration). The migrate operation should offer both pull and push capabilities [11]: components might be pulled from a mashup used by different devices, as well as pushed from the current mashup to a different one.

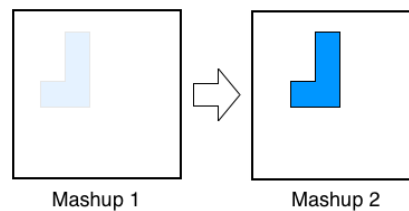


Figure 3.13: Moving components across mashups

Since two different views sharing the same underlying web page influence each other, components need to be internally synchronised. In particular, as described in Daniel et al. [5], we recognise four main abstractions: state, events, responses, and properties. A component should therefore have a state which is shared with other components using the same web page, be able to handle DOM events as well as react to the responses (i.e. updates coming from the web page) and have properties defining information like position and size.

Co-browsing

Distribution implies that components might be moved or copied from a mashup to the other while maintaining state consistency (migrate). On the other hand, *collaborating*, and in particular collaborative browsing, implies that multiple users may work on the same web site in co-operation. In particular, when the same mashup is used by multiple devices, operations produced by one device should be propagated to the others, thus maintaining a synchronised state. Naturally, the same should be valid for the components of shared mashups; and with the simultaneous propagation of interactions, a new series of challenges arise.

Firstly, users should be able to understand what is happening by seeing the results of other users' interactions. For instance, when a user clicks on a button, the others should receive a feedback of the event. The same, of course, should happen for all DOM events and user interactions. Furthermore, the aforementioned mashing and distributing operations should be visible, too. For instance, a moving component might be highlighted, while a notification might identify the user performing the operation.

Another challenge is the communication between users: if devices are used in co-location, users may prefer talking to each other; on the other hand, if devices are used in separate geographical locations, users would need a new communication channel. Instant messaging could be an option [25].

Finally, collaboration gives rise to concurrency problems. These kind of problems are particularly important in editing scenarios, for instance when using text and graphical tools [13], but may also be noticed in operational events: for instance if two users try to move the same component simultaneously in different places.

3.2 Scenarios

To illustrate some possible use cases of a cross-device mashup, we present the following scenarios: Shared Playground, to play online games and puzzles collaboratively, and Living Room News, to share articles and search in aggregated news web sites simultaneously.

3.2.1 Shared Playground

Charlie is a teenager who loves simple online games like Poker, Go and Chess, as well as puzzles like Sudoku and Tic-Tac-Toe. He plays these games and puzzles regularly on his tablet and personal computer: often more than one at a time. However, in order to play different games simultaneously, he needs to continuously switch between different browser windows. Furthermore, he can only save the games into bookmarks which he has to reopen each time. What Charlie needs, is a single page in which he could place his favourite games and only the games themselves, without ads and extra information.

With Jigsaw, Charlie creates a new empty mashup and a number of components encapsulating various game and puzzle web pages: one for Go, one for Chess, one for Sudoku and several for Poker (he likes to play more tables simultaneously). Each created component has a single view showing the game itself, without the extra page elements that are not important.

Instead of managing multiple bookmarks and windows, he can now use a single window with his mashup, and play to his heart's content.

Alice, Charlie's sister, is also quite fond of puzzles even though she is not that good at solving them. In particular, she likes Crossword and Sudoku which she regularly plays on her tablet. She would like to play with her brother, however most of those games and puzzles are either meant for a single device, or for a single player. Furthermore, Charlie's mashup is already full of games and there is no place for Alice to place her favourite Crosswords.

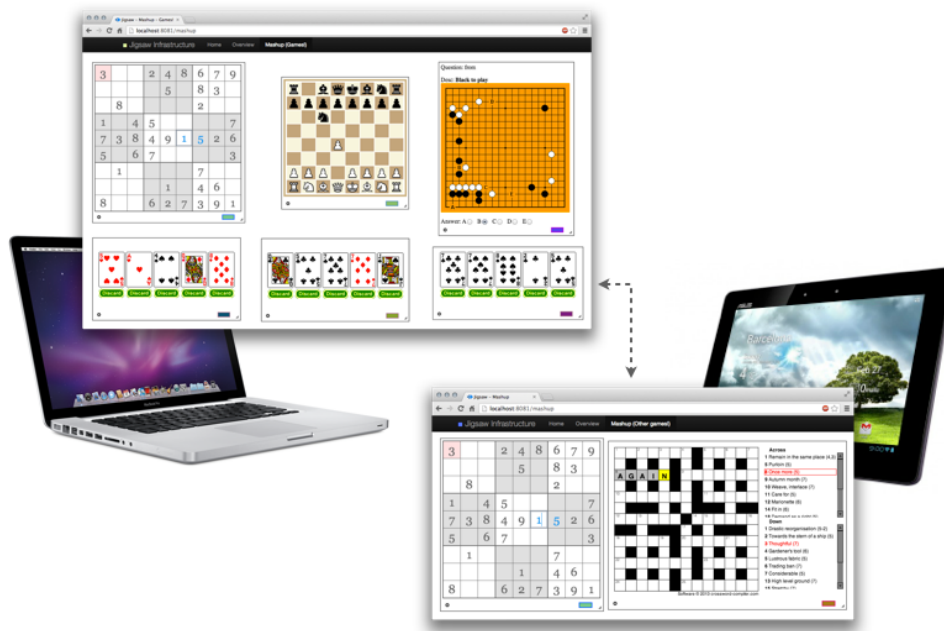


Figure 3.14: Two interconnected mashups used by different devices

She decides to create a new mashup in which she adds her favourite games and puzzles, very much like her brother did. Now she can play with her tablet and, when in trouble, ask Charlie for help. In order to do so, Charlie can either join her mashup and use the component with his sister (collaboration), or pull the puzzle in question into his mashup (migration) and solve the particular problem. When in a hurry, Alice could even push the game into Charlie's mashup. Figure 3.14 shows the two mashups sharing the Sudoku puzzle.

At the end of the day, they have built a shared playground which lets them collect and categorise into various mashups their favourite games and puzzles; solve puzzles that are not meant to be played together (like Sudoku) in a collaborative manner; help each other by migrating components from a mashup to the other; and finally use different devices to access the games from anywhere, even on the go.

3.2.2 Living Room News

A married couple, Malcom and Inara, like to stay informed about current events. However, they are interested in different things: Malcom is quite taken by politics, while Inara is keen on fashion and would like to travel around the world. Their children, Alice and Charlie, who already know Jigsaw, propose them to create a virtual wall of news and articles using the Smart TV in the living room, as well as their tablets.

Malcom creates a mashup with various web sites like the Huffington Post and CNN, each inside its own component. Instead of searching for topics in the three different web sites manually, he links the various search areas together, thus looking for information (e.g. “Obamacare”) in any of the three pages will propagate the operation to the others. Each time he finds an interesting article, he copies the component and creates a view to show only the selected article: the new component will be used as a snippet of the page and eventually sent to the Smart TV (Figure 3.15).



Figure 3.15: Components copied from tablets to the Smart TV

Inara follows Malcom’s example and creates a mashup containing components holding web sites like the National Geographic, World Travel Guide, Flickr and Google Maps. Linking these components together, she can search for a place in any of the web sites and the research query will be propagated to the others. In the case of Google Maps she is only interested in the map itself, therefore she creates a view that only shows the map without search bar and menu.

Both Malcom and Inara can now easily search for topics in multiple web sites simultaneously, as well as sharing and exchanging interesting articles between each other’s tablets and the Smart TV.

4

Jigsaw

In order to realise the presented scenarios, as well as implement many of the proposed requirements, we developed Jigsaw—an infrastructure for cross-device mashups. Jigsaw focuses on direct manipulation of existing web applications to build components, which combined together form mashups that can be accessed by multiple devices in collaboration. Therefore, the infrastructure targets developers, as well as users without programming knowledge, who require a system for rapidly creating cross-device mashups of web applications at the UI layer. Furthermore, Jigsaw takes inspiration from the related works shown in Chapter 2 and in particular from MultiMasher, which describes how components may be extracted from existing web pages and user interactions replayed across devices.

The Jigsaw infrastructure is composed of a back-end and a front-end. The back-end provides the required functionalities to build and manage a cross-device mashup (which may use a number of components encapsulating external web pages) without the need for additional programming or complex modelling. To the user, however, Jigsaw appears as a normal web application and the only requirement for using the infrastructure is a browser, without any additional plug-in. Every user interaction, therefore, from creating mashups to managing components and views, can be executed in the front-end at run-time, with drag and drop operations.

In the following sections we will describe the front-end of Jigsaw using a cross-device mashup as example. This cross-device mashup, depicted in Figure 4.1, is composed of three mashups. The first one is used in a mobile device and for clarity it is simply called “Mobile”. Two components have been placed in this mashup: a search bar extracted from Wikipedia and the grid of a Sudoku extracted from a puzzle web site. The second one, which we called “Research”, uses three web sites: Google Search, Google Images, and Wikipedia. These web sites are linked together in their respective search boxes (i.e. a search made in one web site will be propagated and replayed in the other two). Furthermore, the two components encapsulating Google Images and Wikipedia only show the main content of the pages. The third

mashup, called “Games”, hosts three components: one is using the Sudoku web page while the other two are showing different views of a Tic-Tac-Toe web site.

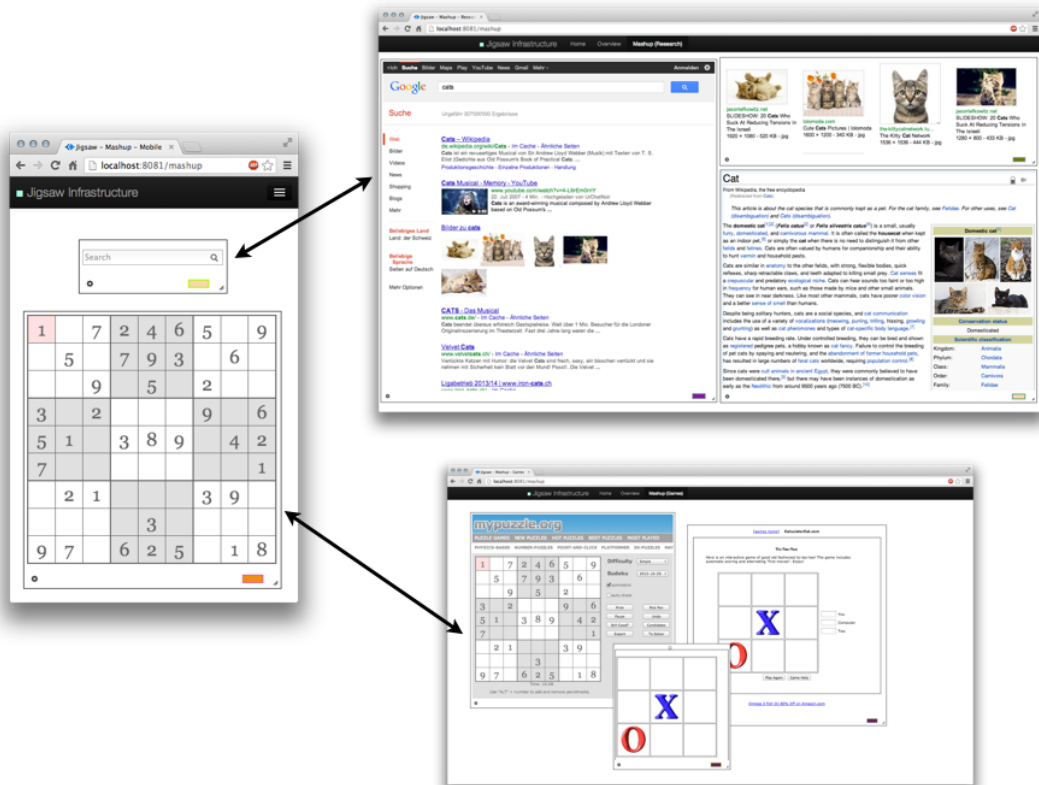


Figure 4.1: Jigsaw, example of cross-device mashup

The same Wikipedia web site is used both in the “Mobile” and “Research” mashups: when the same underlying web site is used by multiple components (cloned), these are automatically synchronised, i.e. all user interaction with one, as well as the DOM changes, are reflected to the others. The same goes for the Sudoku web site which is used in both “Mobile” and “Games” mashups: even though these web elements appear multiple times, the underlying web page is the same and therefore the synchronisation is automatic. For instance, adding a number in the Sudoku on “Mobile” would automatically trigger an update in the Sudoku on “Games”. On the other hand, searches in any one of the web sites in “Research” are propagated to the others: in this case the underlying web pages are different, hence we need to manually link the components. Figure 4.1, for instance shows the results for the search query “cats” in all three pages of “Research”, even though the user originally submitted it to Google Search.

The front-end is composed of three main areas: Home Page, for creating and joining existing mashups; Mashup, for interacting with the components and their web pages; and Overview, for a detailed description of the cross-device mashup as well as for rapidly moving, resizing, and switching components between mashups.

4.1 Home Page

A newly connected user is welcomed in the Home Page, where he can either **create** a new mashup or **join** an existing one. A static gallery of previews shows the mashups composing the cross-device mashup: each preview displays the name of the mashup, a short description, its **components**, and the number of connected **clients** that are using it. As shown in Figure 4.2, components are represented as coloured rectangles: the background colour is defined by the **web page** they are encapsulating, while the border colour represents the current **view** they are showing. Thus, two components that are using the same underlying web page, which are therefore synchronised, can be easily identified.

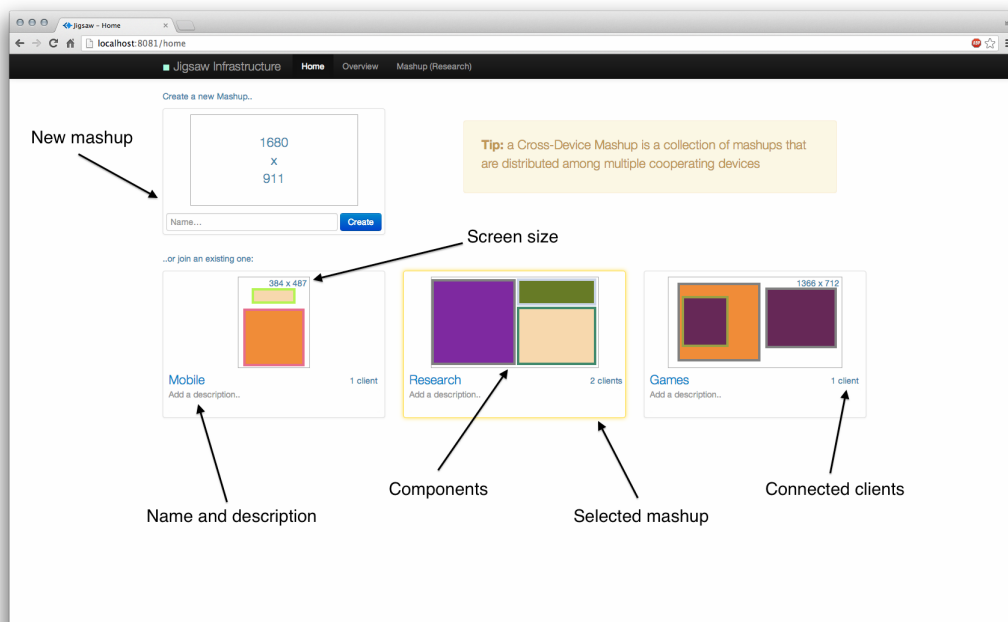


Figure 4.2: Jigsaw, Home section allowing mashup creation and selection

For instance, as it can be seen in Figure 4.2, the user joined the mashup “Research” which is highlighted in yellow: that particular mashup has three components that are using three different web pages. On the other hand, both “Mobile” and “Games” mashups have components with the same background (orange), which means they share the same underlying web page. Furthermore, the border of the two components is different: one is red, the other is grey. Each one, therefore, is showing a different view: grey is the default colour identifying the complete view of the page (i.e. its original appearance, as seen in a normal browser), while red has been dynamically assigned to a custom view, which shows only part of the web page.

4.2 Mashup

The Mashup area is where the user can interact with the selected mashup directly, through the components. Most of the **operations** described in Section 3.1.2, apart from migration of components from a mashup to the other, which is done in the Overview section, are handled here.

Components may be **created**, **destroyed**, **moved** around, **resized**, **copied** and **cloned**. Each operation is propagated to the other connected users immediately, while a notification informs them of the event.

From the components, each showing either a full web page or part of one, the users can interact with the web site itself. For instance, searching something in Google, using Facebook or even playing Sudoku collaboratively. In principle, Jigsaw supports any web site, with some limitations that will be better described later on.

4.2.1 Creating Components

Creating a component is done by selecting an area of the mashup: this will spawn an empty component which, as shown in Figure 4.3, lets the user open either a new web site or an existing one, which will be used by the component. In the latter case, the new component will share the web page with the other components that were already using it, if any.

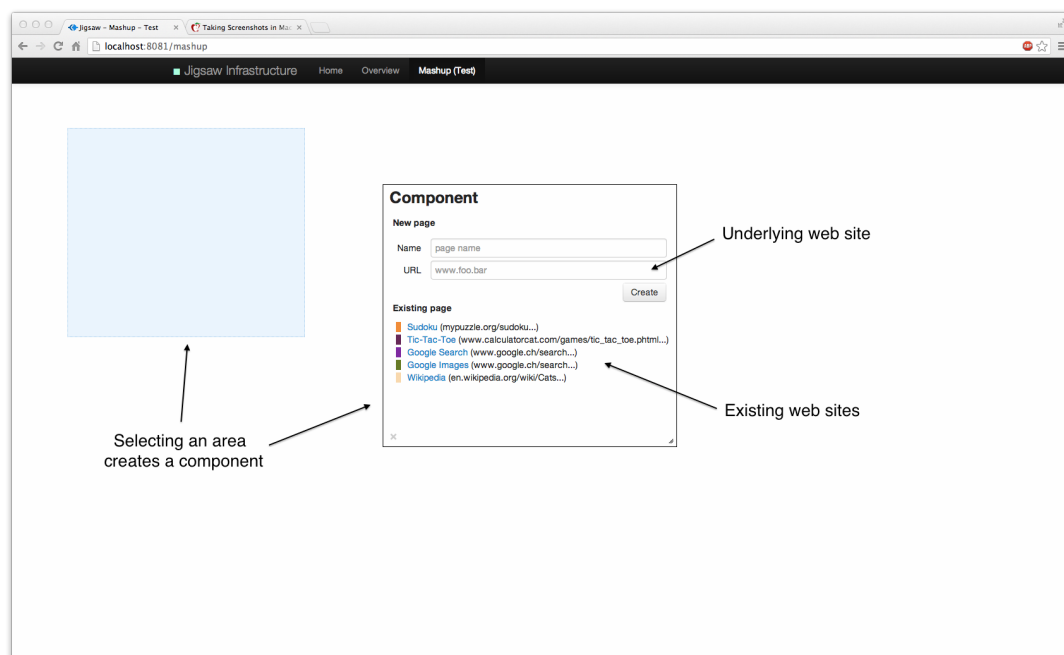


Figure 4.3: Jigsaw, creating components by selecting an area of the mashup

Once the web page has been selected, the component by default shows the document view, i.e. the full page, which might be used like any normal web site.

4.2.2 Component Modes

In order to interact with the components themselves, for instance for **copying**, **cloning**, creating new **views** and changing view; we provide a configuration mode hidden behind the scene. Each component can be switched from *execution mode* to *configuration mode* by flipping it. Figure 4.4 shows both sides of the “Research” mashup components, each of them can be flipped individually by clicking on the bottom left corner.

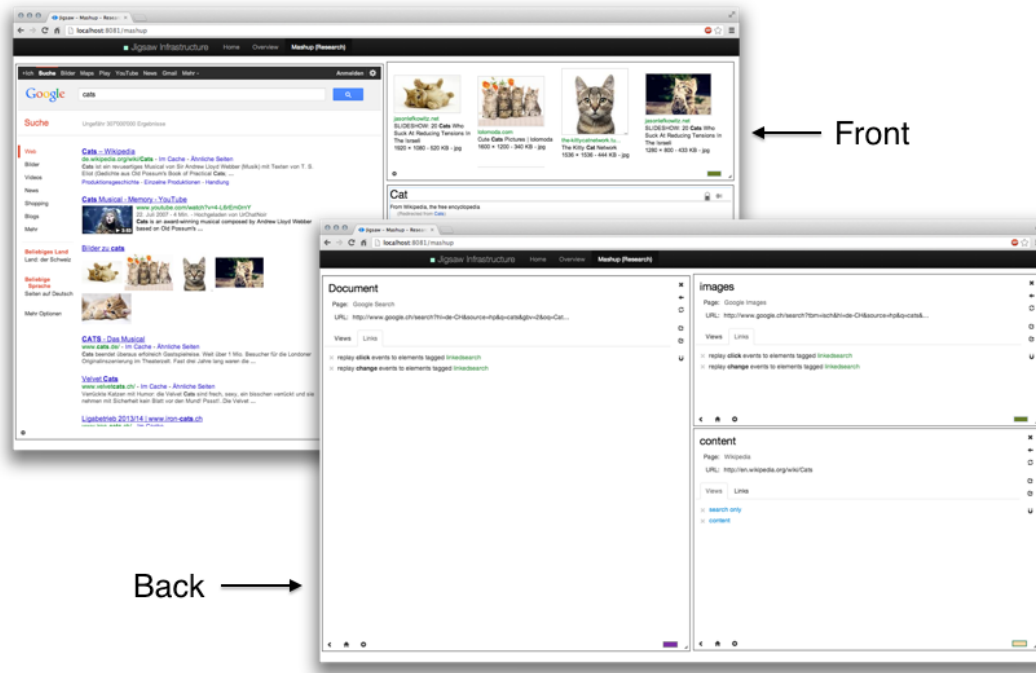


Figure 4.4: Jigsaw, execution mode (front) and configuration mode (back) of components

Components that are flipped over, i.e. that expose their settings, are automatically set into configuration mode, in which both move and resize operations will be reverted once back to normal (execution). This means that a component which is moved in configuration mode, for instance to be handled more comfortably, will automatically return to its original position once the configuration is done.

Moreover, from this modality users might enter two other sub-modes: *view creation mode*, and *link creation mode*.

4.2.3 Component Configuration

To highlight the operations available in such state, Figure 4.5 shows a component exposing the main content of a Wikipedia page.

On the top-left we see the name of the view shown by the component (“content”), as well as the name of the underlying web page (“Wikipedia”) and its URL. The two tabs (“Views”

and “Links”) show the existing views that may be used with the component and the existing links, which are represented by tagged events, used for **inter-connecting components**.

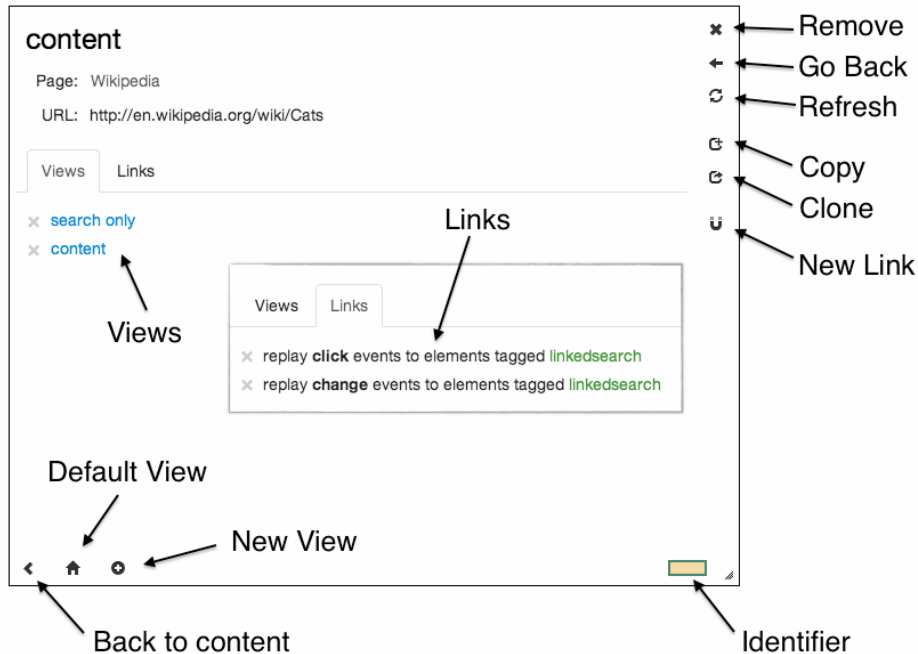


Figure 4.5: Jigsaw, configuration mode (flipped component)

The three buttons in the bottom left corner are used respectively for: flipping the component back, selecting the default view (full document), and **creating a new view** of the web page. The identifier on the bottom right corner is a representation of the component preview detailed in the Overview section.

Finally, the buttons on the right side are divided into three groups: the first one, composed in turn of three buttons, is used to **remove** the component, go back through the underlying web page history, and trigger a refresh event; the second one is used to either **copy** or **clone** the component; and the last button is used for **creating new links** between components.

Creating Views

Once the button to create a new view is pressed, the component enters a yet new mode: *view creation mode*. Any component may be in view creation mode at any time. As shown in Figure 4.6, creating a view takes three steps:

- firstly, the default view of the page is loaded and the user selects the group of DOM elements which should be used (red border);
- then, a preview is shown and, after being named, the view has to be saved;
- finally, the new view can be used with the component.

Furthermore, other components that are using the same underlying web page can now switch to the same view at any time.

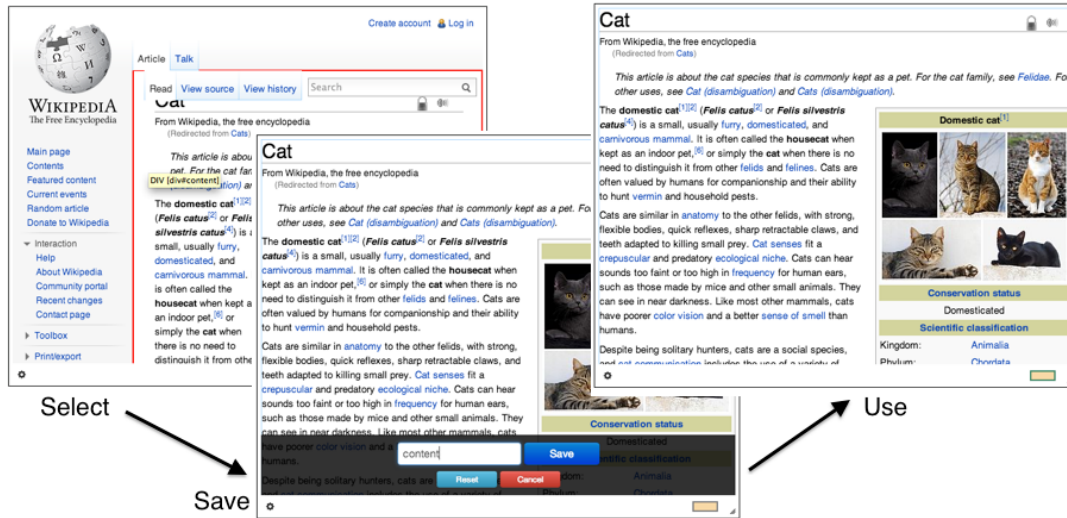


Figure 4.6: Jigsaw, view creation mode

Creating Links

Creating links, on the other hand, requires a more elaborate process. Once the magnet button on the right is pressed, the component enters in *link creation mode*. From there, the user has to start the following procedure:

- specify a DOM event (click, change or submit) which should be automatically replayed to other web pages;
- provide a short text defining the link, i.e. the tag which will be used to perform inter-component communication;
- select the element that should listen to the specified event;
- repeat the procedure for each type of event, in all web pages that need be linked together.

To exemplify this process, let's go back to the scenario: we want to link together Google Search, Google Images, and Wikipedia in a way that if we search something (e.g. "Cats") in any of the three web sites, that search query would be then propagated to the other two, thus synchronising the results.

To do so, from any component using one of the three web sites (it does not matter which one) we initiate the linking procedure:

- we firstly specify the "click" event;
- then, we type as tag "myLinkedSearch"; and

- then, we select the “Search” button, and click Add;
- finally, we repeat the same but with the “change” event, and by selecting the “text input” element.

At this point, we have created a link in the Google Search web site, but no other web site is listening for the same tag (“myLinkedSearch”). Therefore, we repeat the same procedure in the other two web sites. Once done, the linking is complete and the components are inter-connected.

In Figure 4.7, the red border represents a newly selected element, while the dashed borders highlight elements which are either ready to be linked, or that are part of a set of inter-connected components. The colour of the dashed border, furthermore, is based on the specified tag, therefore different groups of linked components will have different colours.

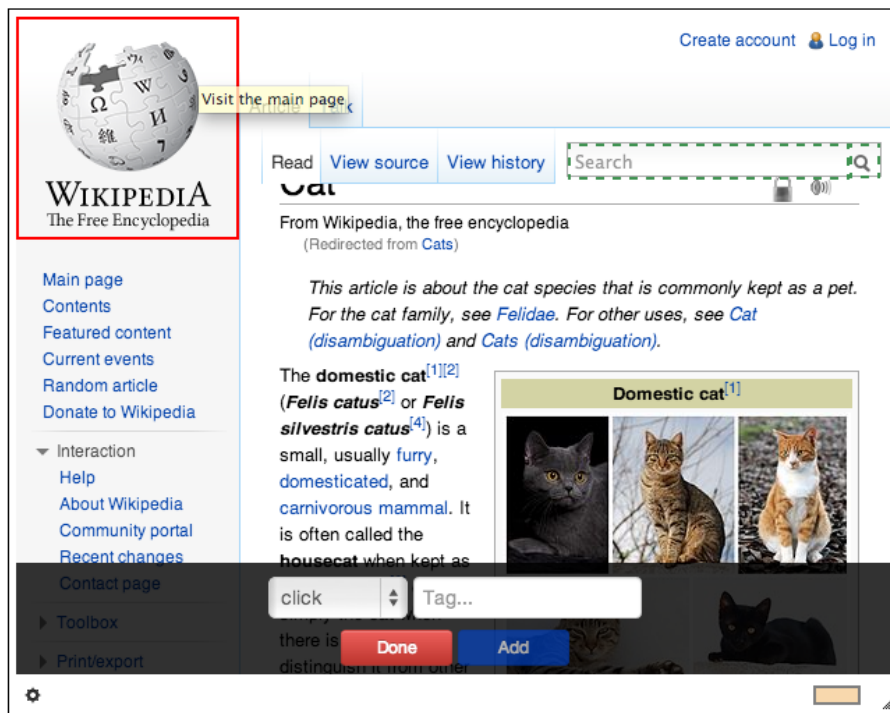


Figure 4.7: Jigsaw, link creation mode

4.2.4 Co-browsing Feedback

As previously described, an infrastructure for cross-device mashups should provide some kind of feedback to let the users know what others are doing. Therefore, we provide a solution based on the usual colour-coding which permeates the whole project.

The results of interactions with components are shown by **highlighting the component** itself with the colour representing the client that initiated the interaction, and by a **notification**

on the bottom right corner as shown in Figure 4.8. Furthermore, user interaction with the DOM elements internal to the web pages themselves are shown by **highlighting the DOM elements** in a similar way.

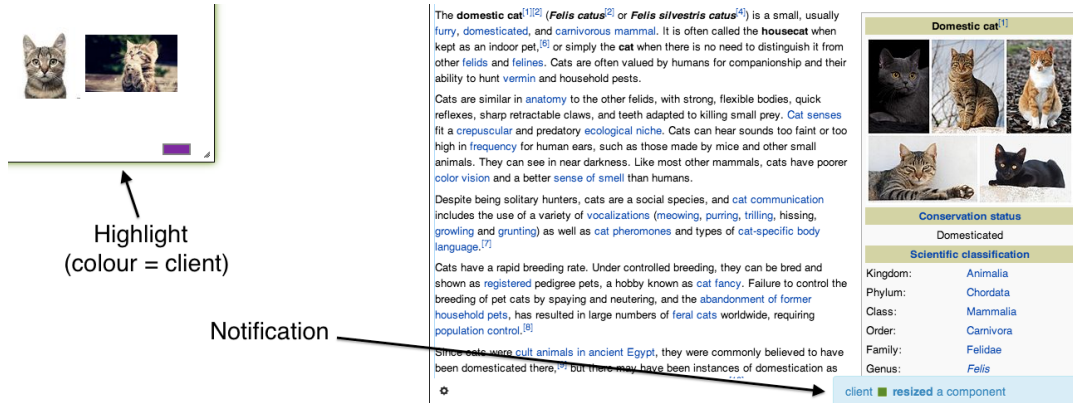


Figure 4.8: Jigsaw, co-browsing feedback while resizing

4.3 Overview

This section of Jigsaw provides a more detailed description of the cross-device mashup, as well as functionalities to manipulate the mashups themselves.

4.3.1 Interactive Previews

First of all, it gives the user some information about how many mashups compose the cross-device mashup and how many clients are using those mashups. Then, it shows a gallery of previews depicting mashups and their components (Figure 4.9), which is very similar to the one presented in the Home section. This gallery, however, is interactive: from here, the user can **move**, **resize**, and **migrate components** from a mashup to the others by using drag and drop actions.



Figure 4.9: Jigsaw, interactive previews

Keeping the collaborative needs presented in Section 3.1.2 in mind, each interaction is instantaneously propagated to the other connected clients. For instance, when a user migrates

a component from a mashup to the other, the following happens: users observing the Overview see the component switching mashup; users in the source mashup see the component disappearing while a notification informs them of the action; and finally users in the receiving mashup see the component appearing, followed by a notification informing them as well.

4.3.2 List of Components

After the preview gallery, the user can see a detailed list describing all the components belonging to the cross-device mashup, as well as the current view of the page they are showing and, if interconnected, the tags defining the connection. The list, better shown in Figure 4.10, is colour-coded following the colours of the components in the preview.

There are 8 components using 5 pages with 6 different views:

Wikipedia (<http://en.wikipedia.org/wiki/C...>) showing search only shares linkedsearch

Sudoku (<http://mypuzzle.org/sudoku>) showing Grid

Google Search (<http://www.google.ch/search?hl...>) showing Document shares linkedsearch

Figure 4.10: Jigsaw, detail of components

The orange component with the red border, for instance, is showing the grid of a Sudoku coming from a puzzle web site, while the purple one is using Google Search and showing the document view (default). Furthermore, this component shares the tag “linkedsearch” with the one using Wikipedia: this tag shows that the two components are interconnected. In order to help the user identify the different components, hovering an item of this list highlights the component in the preview.

4.3.3 List of Clients

In Jigsaw, we focused on clients (devices) and we did not implement the concept of users (there are no user names). However, to show that it would be possible to add it later on, we decided to represent the connected clients with different colours based on their auto-generated session id. The little squares in the gallery previews represent the clients that are using each mashup, while the list at the bottom of the page, highlighted in Figure 4.11, shows all connected clients and the user agent of the devices they are using.

Clients:

- Chrome 30.0.1599 / Mac OS X 10.9.0
- IE 11.0.0 / Windows
- Safari 7.0.0 / Mac OS X 10.9
- Chrome Mobile 30.0.1599 / Android 4.3

Figure 4.11: Jigsaw, detail of clients overview

5

Architecture

One of the most pressing concerns when developing for multi-device and collaborative environments, is session state synchronisation. In Jigsaw, components encapsulate web pages which exist on remote, external servers: these web pages are not developed ad hoc for this purpose, nor are they under the control of the mashup developer. Achieving consistency across devices and inter-connected web pages results therefore even more challenging.

MultiMasher recognises this problem and provides an initial solution based on JavaScript engine input synchronisation which allows DOM events originating from one device, to be replayed to the other devices. To clarify this concept, let us assume two devices are sharing a web site. When the first one clicks on a link, the click event is sent to a server which informs the other device. The latter thereafter replays the event, thus clicking the same link and obtaining synchronisation. Although simple and easily scalable [17], this approach results problematic when consistency is taken into account: firstly, it does not handle updates coming from the web site itself, e.g. AJAX scripts loading data differently depending on the device; secondly, replaying events may produce multiple unwanted updates, e.g. posting something to a forum would create two identical posts coming from the two devices respectively. Moreover, if complex scenarios are taken into account, for instance handling and synchronising multiple different inter-communicating web sites across multiple devices, the problems arising from this solution become even more prevalent.

5.1 State Consistency

It becomes evident that the root of the problem stems from having to maintain a consistent global state which is, however, fragmented and distributed across devices, each with its own session. Since we are handling arbitrary external web sites which are outside our domain, a good approach to this issue would be to centralise the system: instead of trying to synchronise the state of multiple sessions that are distributed among devices, we therefore propose a single session which is in turn shared across devices. To illustrate the difference,

Figure 5.1 compares MultiMasher and Jigsaw approaches: the former uses a server for broadcasting events to the devices, which have different sessions with the same web site; the latter detaches the session from the devices moving it to the server, which becomes a proxy between the devices and the web site.

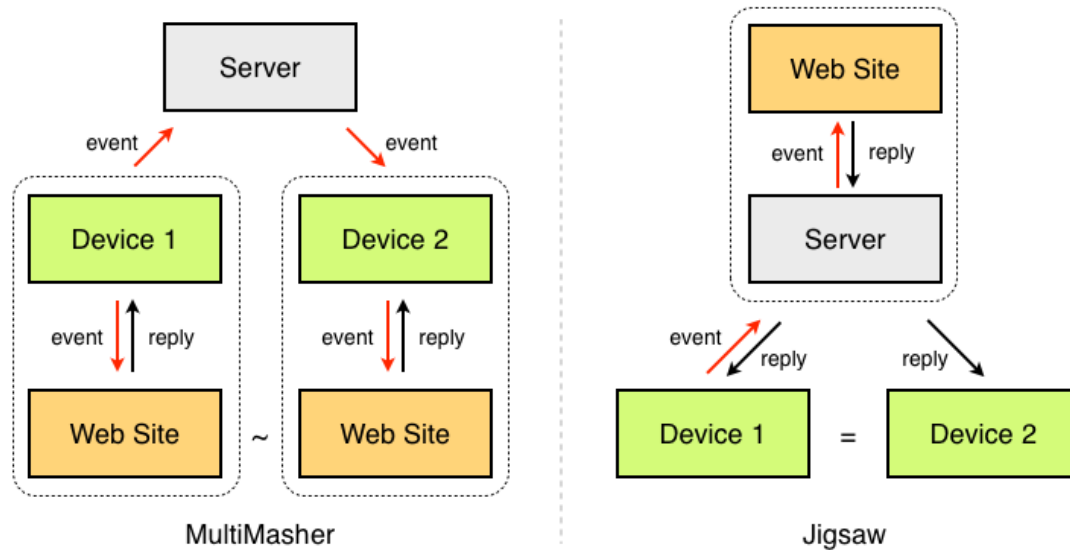


Figure 5.1: Session synchronisation in MultiMasher and Jigsaw

Abstracting from the session state synchronisation of web sites in use, a centralised architecture provides the perfect platform for managing the cross-device mashup entities and operations described in Chapter 3. A cross-device mashup, however, should exist even when no devices are connected: a simple proxy server, therefore, is not enough. In order to solve this issue, we need to completely separate the concept of session between server and external web sites, from the concept of session between connected devices and server. We propose therefore a remote control architecture, which gives the proxy server a dual role: from the point of view of the devices, it serves as a normal proxy; from the point of view of the web sites, however, the server is seen as a browser which is requesting resources even if no user is actually using it.

5.2 Remote Control

The Jigsaw infrastructure uses a remote control architecture conceptually similar to those described in Section 2.4. This means that the back-end, at least when interactions with web pages are concerned, has to act as an independent browser: in Highlight [21] they actually use a real browser; while in Cheng [2] the author conceptualises the creation of a virtual browser from scratch. In Jigsaw we decided to use a particular kind of browser: a headless browser. A headless browser, simply put, is a web browser without a graphical user interface¹. In particular, we use the headless browser in conjunction with our server logic: the

¹<http://blog.arhg.net/2009/10/what-is-headless-browser.html>

former handles remote web pages (HTTP requests and responses) and third-party resources in general (included remote JS execution); while the latter abstracts from the concept of web pages and handles the Jigsaw infrastructure itself, e.g. mashups, components, and user interactions. Clients connected to the infrastructure therefore control the server logic, which in turn uses the headless browser to handle remote web pages as shown in Figure 5.2.

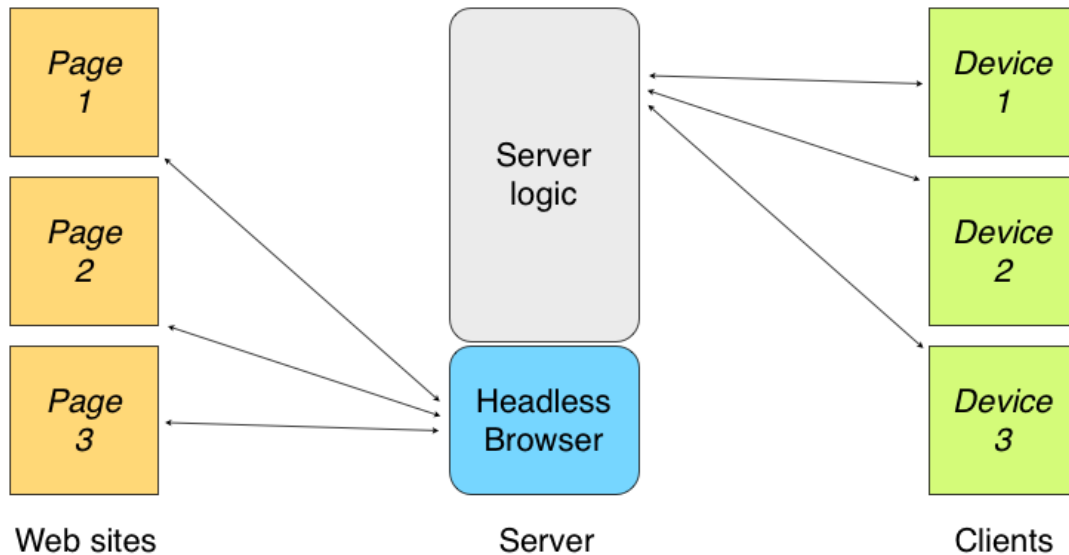


Figure 5.2: Architecture, overview of the infrastructure

As previously mentioned, this infrastructure allows us to completely separate the sessions between external web sites and the headless browser, from the local sessions between clients and server logic. Since the headless browser acts as unique client to the external web sites, we do not need to manage those sessions: they are not shared and cannot get out of sync. On the other hand, we need to manage the local sessions between clients and server logic: instead of managing complex interactions between devices, like those that would exist with a peer-to-peer architecture, we decided to give the server a central role.

5.3 Server Logic

The server logic is modular and handles both the headless browser on one side, and the connected devices on the other.

5.3.1 Headless Browser

The headless browser opens and executes web pages like a normal browser would. It can, furthermore, communicate with the server logic: in particular, it can notify it when a web page finishes loading, or resources have been received. The server logic, in turn, can fire DOM events like clicks, changes and submits to pages that are open inside the headless browser, and it can ask for the DOM of a page as seen in Figure 5.3.

With some limitations (e.g. DOM changes cannot be immediately reported) the server logic can control the headless browser as if it were a normal browser with multiple tabs open: new tabs can be created, existing tabs can be destroyed, pages can go back and forward in their history, and they can even be refreshed.

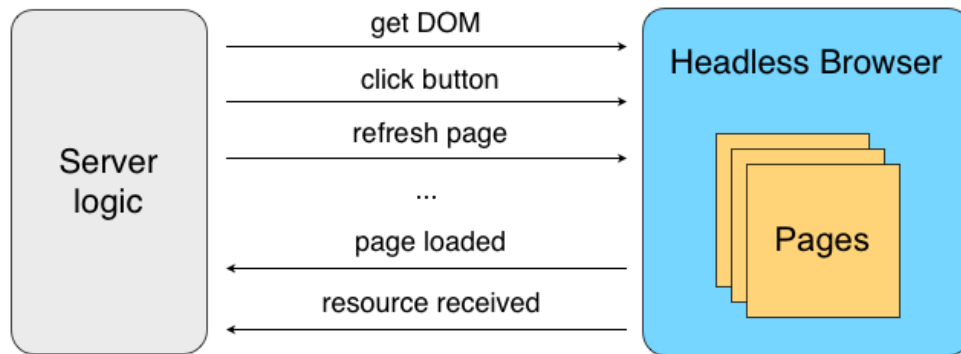


Figure 5.3: Architecture, headless browser

Jigsaw, however, does not deal with web sites directly, but through components that are inter-connected and placed inside mashups. The infrastructure, therefore, needs to abstract from the concept of web page and instead model the elements and operations described in Chapter 3. In particular, as shown in Figure 5.4, the page logic extracts the DOM of web pages from the headless browser using the browser logic. Once extracted, the DOM is transformed into multiple views defined by the users, and finally sent to the interested clients.

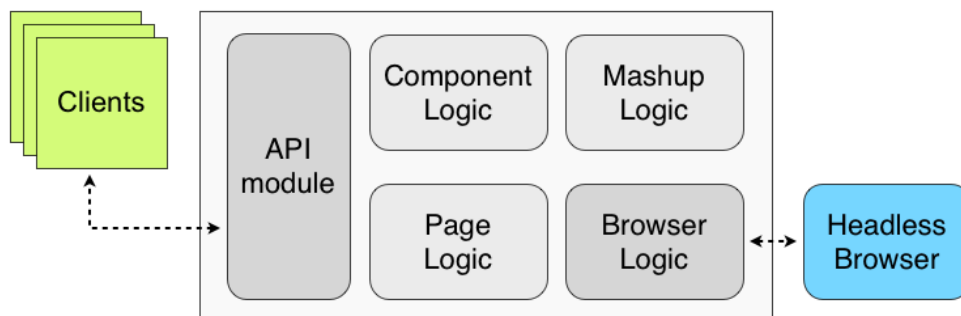


Figure 5.4: Architecture, server logic

5.3.2 Clients

The clients interact with the server logic using a set of predefined APIs which describe the available operations. Operations that deal with components are then handled by the component logic, operations that deal with mashups by the mashup logic, and so on. For instance, when a client moves a component the following happens: using the proper API, the client send the new component position to the server; the API module on the server recognises the component operation; and it tells the component logic to perform it. Once the

operation is complete, the component logic uses the API module to push the result to all the listening clients (Figure 5.5).

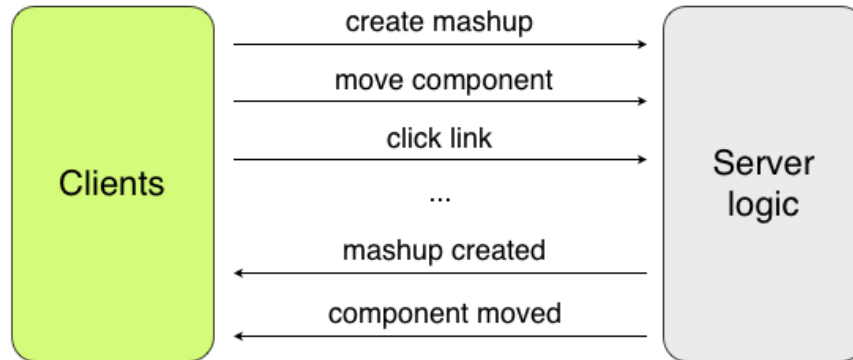


Figure 5.5: Architecture, API module

Each module defined in the server logic takes care of a different aspect of the cross-device mashup:

API module As previously described, the API module connects the clients using the infrastructure with the inner logic of the server. API module and browser logic are at the core of the remote control metaphor used by Jigsaw: clients use the APIs to control the cross-device mashup elements and perform the necessary operations; the API module interacts with the other module in the server logic to access and control the headless browser.

Component Logic The component logic handles operations that directly impact a component. These operations include move, resize, view selection (each component shows a particular view of the underlying web page), and page creation. Furthermore, a component may ask for the source code of the current view which it is displaying.

Mashup Logic Operations that directly modify mashups are handled by the mashup logic. Here we can create, modify, join, destroy, resize, and request mashups with all their components. Furthermore, this module deals with adding, removing, cloning, copying, and switching components between mashups.

Page Logic Each representation of a web page points to a URL, has a description, and provides a set of views which show various DOM elements of that page, as well as a set of endpoints describing links between different inter-communicating web pages. The page logic, therefore, handles operations like view and link creation and management. Furthermore, it handles user interactions with the web pages themselves, that are coming from the API module. For instance clicks, changes, submits, and page refresh.

Browser Logic Finally, the browser logic interacts with the headless browser through a simple set of internal API. It can open and close tabs, load web pages, simulate user interactions (e.g. clicks), and it can receive a web page DOM either by directly requesting it (e.g. after a change event), or automatically when either a page finishes loading or new resources are received.

5.4 Data Model

The server logic provides persistence through the use of a database: each cross-device mashup element is therefore modelled as a persistent document. In particular, we identify three main models: Mashup; Component; and Page. Figure 5.6 shows a more detailed diagram representing the entity relationships.

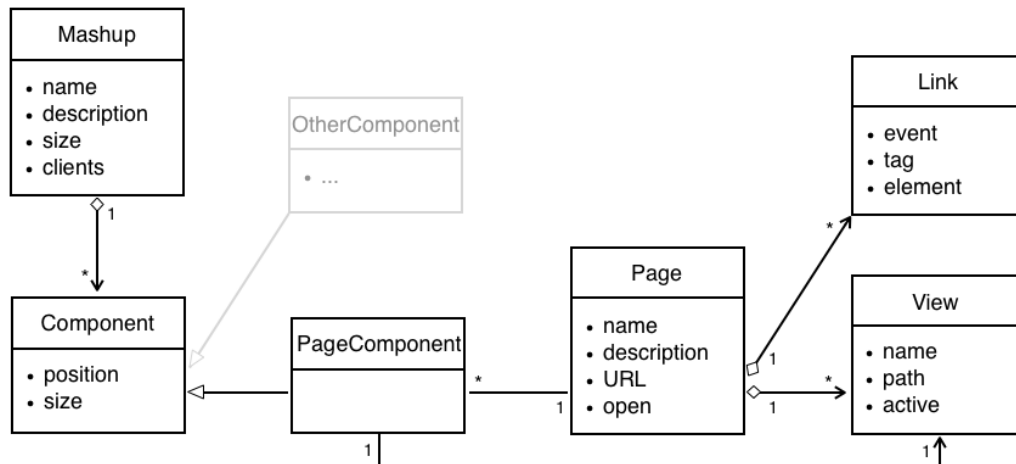


Figure 5.6: Architecture, data model

A Mashup has a name, a description, a size describing the resolution used by the connected device, and a list of clients that are using it. Furthermore, each Mashup may have Components.

A Component is an abstraction defined by position and size: the data model is easily extendible and in the future could include various kind of components (e.g. local components used to add notes, or additional content which is not taken from a web page). In Jigsaw, we implemented only one kind of components: PageComponents. Each PageComponent shows a particular View of a Page, and multiple PageComponents may use the same Page.

A Page has a name, a description, an URL, and a flag defining if the web page is open inside the headless browser. Furthermore, a Page has a collection of Views and Links. The former have a name and a CSS path pointing to the DOM tree displayed by the View; while the latter are defined by a tag, an element and an event which will be replayed to other Pages using the same tag-event pair (e.g. “click event” on “button element” tagged as “MyLinkedException”).

5.5 Thin Client

Using a remote control architecture means creating a thin client [26] which relies heavily on back-end operations. The browser itself, at least in its simplest form, may be seen as a thin client. The Jigsaw client, in particular, initiates server operations and displays the results without computing any complex computation locally. The front-end features provided by Jigsaw are described in Chapter 4: it is composed of three main sections (Home, Overview

and Mashup), each used at different stages of the cross-device mashup manipulation.

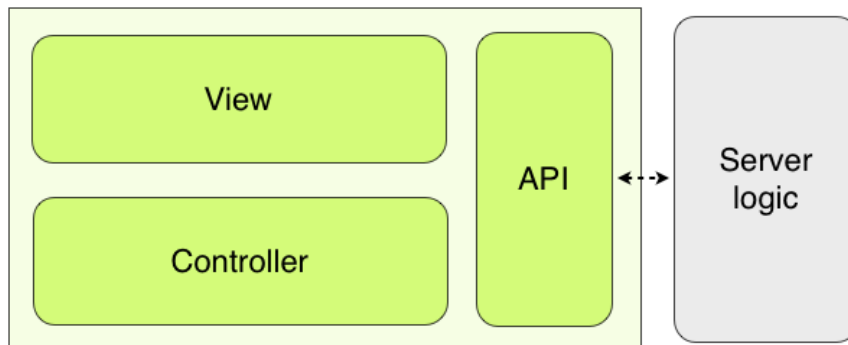


Figure 5.7: Architecture, thin client

In general, we can describe Jigsaw as a Model-View-Controller (MVC) architecture where the Model is represented by the server, the Controller is split between server and client, and the View is displayed in the front-end as shown in Figure 5.7. In particular, user interactions with the View are handled by the Controller which, using the proper API, contacts the server logic and waits for results to later display in the View.

6

Implementation

Jigsaw is a web application, the presented infrastructure therefore has been developed using web technologies. Taking inspiration from MultiMasher, and more precisely from the tool developed as part of that paper (*distribution.js*), we decided to build the entire infrastructure, both server and client-side, using state-of-the-art JavaScript technologies.

Douglas Crockford, one of the most prominent programmers involved in the development of the JavaScript language defines it as the lingua franca of the Web¹, this is evidently true when talking about front-end applications running inside browsers: almost any browser on any device, in fact, supports JavaScript and there are plenty of JavaScript front-end frameworks. However, in recent years, especially after the advent of Node.js² in 2009, JavaScript is becoming ever more popular as back-end programming language [24].

In this section, we will introduce these technologies, as well as describe the implementation of the Jigsaw architecture in detail.

6.1 Technologies

Client and server in Jigsaw communicate via a series of APIs implemented as WebSocket messages³. WebSockets have been introduced in HTML5 in order to provide a protocol for two-way communication with a remote host, directly from the browser. Jigsaw operations, therefore, are asynchronous by nature: each message from the client is parsed by the server and transformed into an operation. When a response is ready, the server creates a new message and pushes it back to the client.

¹<http://javascript.crockford.com>

²<http://nodejs.org>

³<http://www.websocket.org>

Since we used a number of different technologies in both server and clients, we will discuss both sides separately.

6.1.1 Server-side

The server is divided into three main services depicted in Figure 6.1: a database, to store the data model and keep persistence; an headless browser, to open and interact with remote web sites; and finally a back-end JavaScript platform that provides a web server as well as an abstraction of WebSockets, and which executes Jigsaw's server logic.

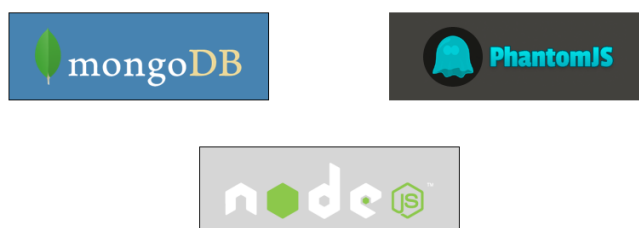


Figure 6.1: Implementation, main server-side technologies

As database, we use MongoDB, which is an open-source document database, and the leading NoSQL database⁴. Even though it provides many interesting features like replication, high availability, and auto-sharding; such advanced features are not needed in this project: we decided to use MongoDB because of its document-oriented storage architecture that uses JSON-style documents, which are ideal in a JavaScript environment.

For the headless browser, two solutions were available: ZombieJS⁵ and PhantomJS⁶. Both tools are mostly used for page automations and testing of local JavaScript rich web pages. ZombieJS is presented by the authors as a lightweight, full-stack testing utility which emulates a JavaScript environment. Furthermore, it is distributed as a module for the back-end platform we are using. However, it is quite limited and does not provide any real rendering of DOM elements. PhantomJS, on the other, is a full-stack headless browser which uses WebKit. It provides, therefore, fast and native support for various web standards: DOM handling, CSS selector, JSON, Canvas, and SVG. It is, however, more complex than ZombieJS and requires a separate process, instead of being developed as a module for the back-end platform. After some testing, we decided to use the more complete, although more complex, solution in order to support a range of web sites as wide as possible: PhantomJS.

The back-end JavaScript platform that executes the logic of Jigsaw is Node.js: a framework for developing high-performance, concurrent programs that don't rely on a multi threading approach, but use asynchronous I/O with an event-driven programming model. Node.js provides a number of modules, or libraries, which greatly extend the capabilities offered by the platform.

⁴<http://www.mongodb.org>

⁵<http://zombie.labnotes.org>

⁶<http://phantomjs.org>

In particular we use the following modules:

Mongoose ⁷ This module bridges Node.js and MongoDB, by providing a straight-forward, schema-based solution to modelling the application data as JSON objects.

Phantom ⁸ To bridge Node.js and PhantomJS, this module offers inter-process communication, allowing Jigsaw's server logic to control the web pages that are open in the headless browser.

Cheerio ⁹ Before being sent to a client, the DOM of a web page has to be modified on the server: either for safety and functional reasons, or for creating different views of the page. This module allows fast manipulation of DOM trees, implementing methods similar to those provided by jQuery.

Socket.io ¹⁰ This module aims to make real time applications possible in every browser and mobile device, blurring the differences between the different transport mechanisms. If available, it uses HTML5 WebSockets, otherwise it falls back to using other mechanisms like AJAX long-polling.

Express ¹¹ To help serving local content to the clients (the three pages comping Jigsaw's front-end), we use the Express module, which is a minimal and flexible web application framework.

Jade ¹² Instead of writing HTML directly, the front-end pages' structure is written using the Jade template language, which is then compiled by Node.js into standard HTML5.

Should ¹³ To help creating a series of tests to verify the database integration, we used Should. This module provides an expressive, readable, test framework agnostic, assertion library for Node.js.

UserAgent ¹⁴ Finally, in order to show nicely formatted user agents (i.e. browser/device) in the front-end Overview section, we use the UserAgent module.

Furthermore, as JavaScript IDE and HTML editor we used WebStorm¹⁵. This IDE provides a number of useful features: complete integration with the Node.js platform, database inspector for MongoDB, SVN client for rapid synchronisation and backup, and an intelligent JavaScript auto-completion mechanism similar to the one provided by the Eclipse IDE for Java.

⁷<http://mongoosejs.com>

⁸<https://github.com/sgentle/phantomjs-node>

⁹<https://github.com/MatthewMueller/cheerio>

¹⁰<http://socket.io>

¹¹<http://expressjs.com>

¹²<http://jade-lang.com>

¹³<https://github.com/visionmedia/should.js>

¹⁴<https://github.com/3rd-Eden/useragent>

¹⁵<http://www.jetbrains.com/webstorm>

6.1.2 Client-side

The role of the Jigsaw client is dual: it is used to create, manage, and use the cross-device mashup, as well as to show broadcast information about the cross-device mashup to the connected users. In order to simplify collaboration, a user's interactions have to be clearly and immediately propagated to the others. Therefore, we decided to use a framework that could facilitate immediate updates of web page elements, while supporting a two-way, push message, mechanism. As shown in Figure 6.2, we use the open-source MVC JavaScript framework by Google: AngularJS¹⁶.



Figure 6.2: Implementation, main client-side technologies

AngularJS nicely integrates with the client side library of the aforementioned Socket.io, hence the two-way communication channel is guaranteed to work even in browsers that do not support WebSockets natively. The main idea behind AngularJS, as in other MVC frameworks, is the separation of Model, View, and Controller. The Model is represented by the data coming from the server logic via Socket.io. The Controller parses this data and defines the behaviour of the various elements composing the cross-device mashup (e.g. the view creation procedure and the components' behaviour itself). The View, finally, is bound to the model data parsed in the controller: when the data changes, the View automatically changes to represent the updated data. For instance, if an array of elements is represented as a HTML list, and one of these elements is removed from the model, the HTML list automatically adapts to show the updated array. Furthermore, AngularJS supports the creation of custom HTML elements defined as directives. Each mashup component, for instance, is represented as a directive.

To design a pleasant and vastly recognisable UI, we use HTML and CSS-based templates provided by the Twitter Bootstrap¹⁷ framework for typography, forms, buttons, navigation and other interface components. All the icons shown in the components' configuration mode, for instance, come from Bootstrap.

Additionally, we use jQuery and jQueryUI¹⁸ as helpers for easily selecting and capturing events in DOM elements (for instance user interactions with the web pages displayed inside the components), as well as for implementing drag and drop operations like moving and resizing components.

Finally, we use a number of little JavaScript libraries for various needs:

¹⁶<http://angularjs.org>

¹⁷<http://getbootstrap.com>

¹⁸<http://jquery.com>

Shadow Animation ¹⁹ A jQuery plugin that we use for showing collaborative interactions by highlighting components with the colour of the client that initiated the operation.

Twinkle ²⁰ This library creates various coloured effects in specified positions of the page. Twinkle is used to highlight collaboration internal to web pages (clicks and changes).

UI Touch Punch ²¹ A plugin for jQueryUI that adds Touch Events support to buttons, as well as draggable and resizable elements.

Sugar ²² A JavaScript library which extends native objects to help handling arrays and other JavaScript structures.

Alertify ²³ In conjunction with Bootstrap, this library shows pleasant notifications in the bottom-right corner of the application.

MD5 ²⁴ In order to create dynamic colours based on objects' IDs, we take the first 6 characters of the hashes computed by this library.

6.2 Communication

Taken as a whole, the Jigsaw infrastructure connects clients and server through asynchronous two-way messages, and server to external web sites through the usual HTTP protocol. Messages are implemented using facilities provided by Socket.io, which leverage the WebSocket protocol when available. To communicate with browsers that do not support WebSockets, Socket.io uses a number of fall-back mechanisms like, for instance, AJAX long polling. The benefit of using messages through WebSockets instead of the most common XMLHttpRequest (XHR) mechanism, is that with messages the clients do not have to continuously poll the server for updates, i.e. the server is able to push messages to the clients directly. Furthermore, this two-way channel between client and server is more efficient than XHR. In fact, HTTP introduces an overhead that is not present in WebSockets: additional header information is sent upon request and each subsequent request. The headless browser inside the server acts as a real browser, therefore it uses the standard HTTP protocol to interact with external web pages.

6.2.1 Protocols

To illustrate the two communication protocols used by the infrastructure, Figure 6.3 presents a general example scenario. A cross-device mashup formed by a number of mashups, each hosting various components with different views, uses two web sites:

- both web sites have been linked together through their search bar, i.e. a search query in one web site is replayed to the other;

¹⁹<http://www.bitstorm.org/jquery/shadow-animation>

²⁰<http://larsjung.de/twinkle>

²¹<http://touchpunch.furf.com>

²²<http://sugarjs.com>

²³<http://fabien-d.github.io/alertify.js>

²⁴<https://github.com/blueimp/JavaScript-MD5>

- two devices are connected to the infrastructure and are using different mashups;
- the first mashup uses one component showing the full view of one web site (Google Search);
- the second mashup has two components showing only the main content of both web sites (search results in Google Search and Wikipedia).

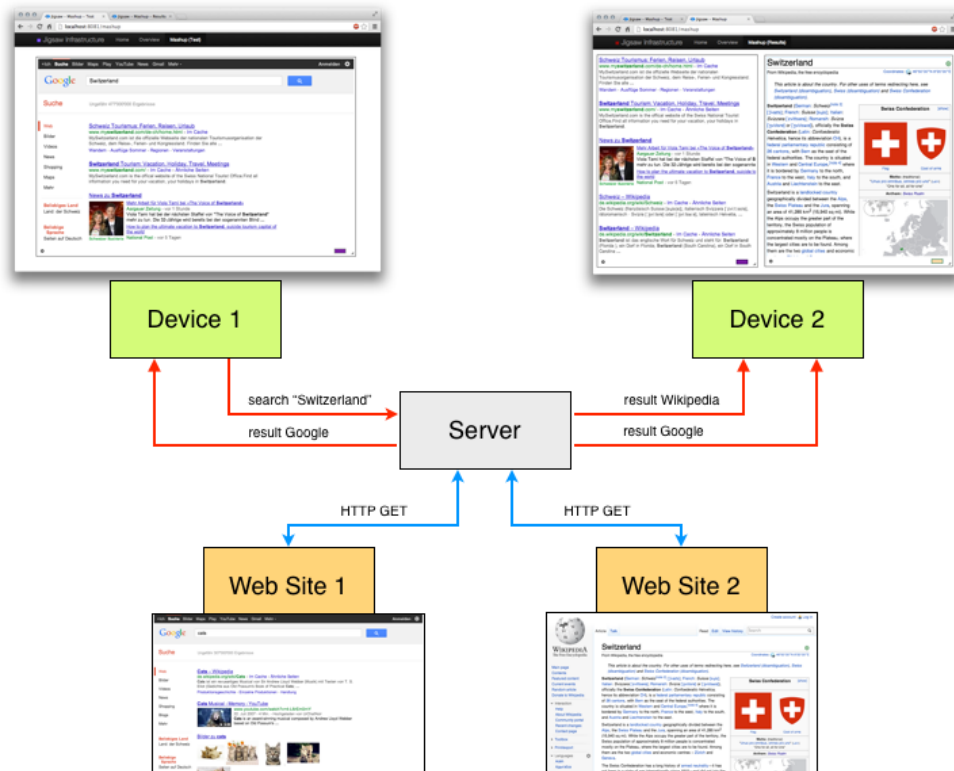


Figure 6.3: Implementation, infrastructural overview, scenario

Device 1 searches “Switzerland” in the component showing Google Search: the change event is sent to the server as WebSocket message and, once the search form’s submit event is fired, the server (through PhantomJS) uses the HTTP protocol to fetch the result. Since the two pages have been linked together, upon receiving the search query for Google Search, the server logic propagates it to Wikipedia, thus fetching the results from that web site, too. At this point, both underlying web pages in the headless browser are showing results for the query “Switzerland”.

The server logic now extracts the DOM of the full Google Search web page and sends it to Device 1. From the full DOM it creates a view showing only the content (search results) and sends it to Device 2. Finally, it extracts the DOM from Wikipedia, content only, and sends it to Device 2 as well. The red arrows in Figure 6.3 depict WebSocket messages, while the blue arrows refer to the HTTP protocol.

6.2.2 Events

Clients and server communicate through a custom API based on Socket messages: since we are using Socket.io, we define the format of such API as “event, data”. The following list summarises the available events, separated by area of interest:

Mashup: create, load, destroy, resize, updateInfo, getList, getAllMashups, addComponent, addPageComponent, removeComponent, cloneComponent, copyComponent, switch-Component;

Component: move, resize, createPage, setPage, selectPageView, getPageSource

Page: addView, removeView, addLink, removeLink, updateInfo, updateViewInfo, getList, change, click, submit, goBack, goForward, reload

As can be deduced from this list of events, every operation initiated by the client is executed in the server and propagated to the other clients when necessary. Everything is done by asynchronous socket messages, therefore after each event has been executed, a new message is sent back as response. For instance, when a client uses the “component:move” event, the server moves the component and, only when the operation is complete, broadcasts a new “component:move” message to the all the interested clients that will thereafter update their view. Furthermore, in order to broadcast messages to only the interested clients, we use a publish/subscribe mechanism detailed later in this section. For instance, if two clients are sharing the same mashup, only these two clients will be interested in receiving updates related to that mashup.

Server API

The server receives events from the clients (inbound) and it initiates the respective server operations. Then, it takes the results of these operations and it broadcasts them back to the clients (outbound). Since we are using Socket.io, each client is represented as a socket: the server maintains a list of all the connected sockets which is used to broadcast messages. This means that there is no P2P functionality, i.e. user interactions initiated in a device (using a socket), are sent to the server which then decides what sockets need to be informed.

The API module is divided into five files representing different categories of events: handshake, mashup, component, page, and browser.

Handshake This file provides handshake functionality by observing socket connect and disconnect events. Furthermore, here we implemented some debug logic to show the underlying Socket.io messages that are exchanged between clients and server.

Mashup Here we take care of the API that directly modifies a mashup, for instance it is able to retrieve a list of all the mashups and sent it to the client.

Component Events that directly impact components are managed by this file, for instance a resize operation.

Page This file manages API related to the web pages encapsulated into PhantomJS (the headless browser), like clicks, but also events related to the creation of views.

Browser Finally, here we manage outbound events coming from the headless browser through the browser logic. In particular we use this file to send a web page source (the DOM used by a certain view), the web page URL (which might change and has to be kept synchronised), and linking information (confirming linked actions between different web pages).

In order to propagate events through messages to the interested clients, each socket might be registered to a number of rooms. Rooms define a publish/subscribe mechanism: sockets might join a room, leave a room, and be registered to multiple different rooms at the same time. When a message is sent to a particular room, each client who joined that room receives the message. In practice we can explain this mechanism with a simple example outlined in Figure 6.4: two clients are sharing the same mashup; the first client removes a component from that mashup; the second client sees the component disappearing.

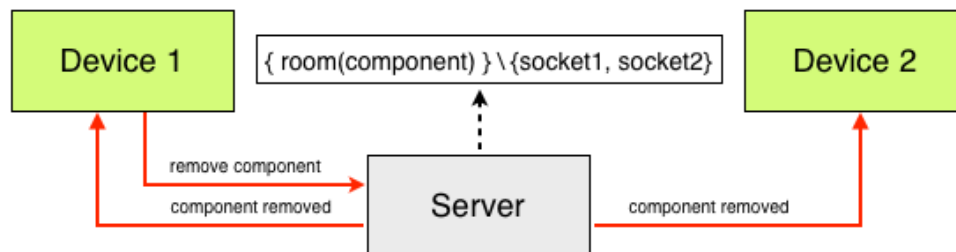


Figure 6.4: Implementation, rooms

Behind the scenes, the server is managing a number of rooms that both sockets have joined:

- the mashup room, defined by the mashup ID;
- a room for each component, defined by the component ID;
- a room for each web page, defined by the web page ID;
- and a room for each view, defined by the view ID.

Each room takes care of a different aspect of the cross-device mashup, for instance the clients are registered to the same mashup room, therefore events that modify that particular mashup are broadcasted to both clients.

In this particular case, when the component is removed, the clients receive a confirmation of the event and then they get removed from the component room, as shown in Listing 6.1. Thus they will not see updates relative to that component in the future.

```

1 var mashupEvents = function(io, socket, mashupLogic) {
  socket.on('mashup:removeComponent', removeComponent)
  /**
   * Removes a component from the mashup
   * @param {{ mashupId, compId }} data
   */
6  function removeComponent(data) {
    if (data && data.mashupId && data.compId) {
      mashupLogic.removeComponent(data.mashupId, data.compId)
    }
  }
}

```

```

11      , function(component) { //success
      //remove all sockets
      var compClients = io.sockets.clients(data.compId)
      compClients.forEach(function(client) {
        client.leave(data.compId)
      })
16      //broadcast success
      emitComponentRemoved(data.mashupId, component)
    }
    , function(err) { emitError(err) } //failure
  )
21  }
}
/**
 * Broadcast mashup component added
 * @param mashupId
26  * @param {String} compId
 */
function emitComponentRemoved(mashupId, compId) {
  socket.broadcast.emit("mashup:removeComponent", {
    mashupId: mashupId
31    , compId: compId
    , client: socket.id
  })
}
}

```

Listing 6.1: API example, removing a component

Finally, the API implemented in Jigsaw can be quickly extended and modified thanks to its modularity which abstracts from the server logic itself. Taking advantage of Node.js, in fact, everything is asynchronous.

Therefore, the general work flow when managing a cross-device mashup is depicted in Figure 6.5: upon receiving a message, the server identifies the event and, from the API, it calls the appropriate callback function implemented in the server logic. The server logic, in turn, executes operations using documents extracted from the database. The operation in the server logic firstly updates the model, then executes a callback which gets to the API, where finally a new message is formed and sent to the proper rooms.

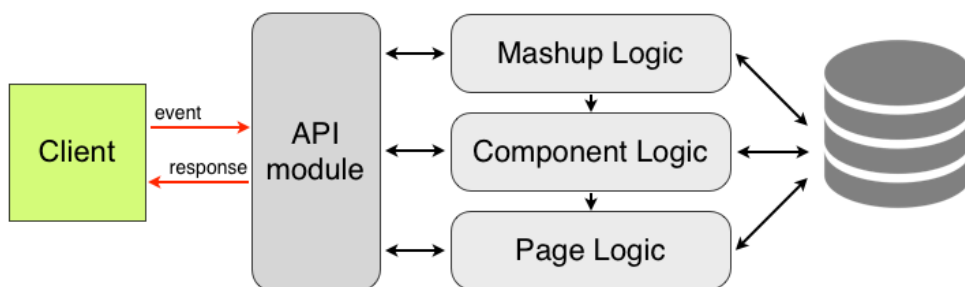


Figure 6.5: Implementation, general work flow

Client API

This API is handled by the AngularJS controllers using a client-side library for Socket.io. Therefore, each API implemented in the server has a counterpart in the client. Not every controller needs all the API methods, therefore each controller only implements a subset. For instance, the MainController, which is the only application-wide controller, only implements the initialisation handshake, error messages, and mashup creation and load events.

Listing 6.2 shows a snippet of code extracted from this controller: events are registered in “socket.forward” and implemented later on, for instance in case of errors the client simply shows a notification.

```

socket.forward([
  'error'
  , 'mashup:create'
  , 'mashup:load'
5  , 'init'
], $scope)

//error notification
$scope.$on('socket:error', function (ev, data) {
10   alertify.error("<strong>Error</strong> (" +data.type+"):
      "+data.message);
})

```

Listing 6.2: Example of front-end API implementation

6.3 Server

The back-end is modelled around the concept of remote architecture defined in Section 5 as well as the data model described in Section 5.4. As can be seen from Figure 6.6 we define five central modules: API, Mashup Logic, Component Logic, Page Logic, and Browser Logic.

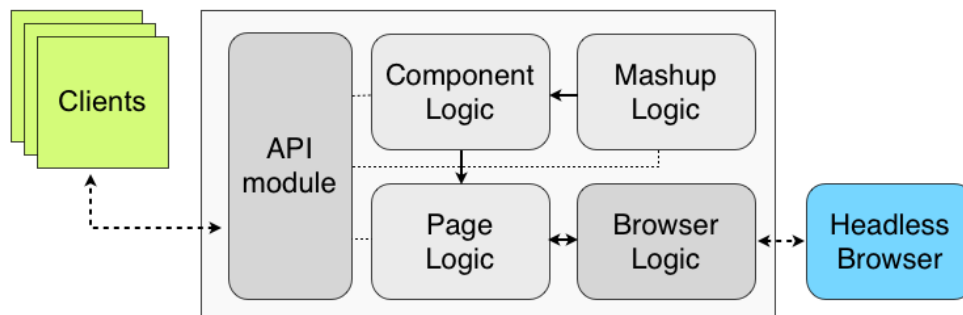


Figure 6.6: Implementation, back-end

6.3.1 Mashup Logic

The mashup logic is responsible for managing mashups as well as creating, removing, and migrating components. Components, in fact, as seen in Section 5.4, cannot exist without

mashups. Using the “Mashup” model, which Mongoose extracts from the database, this module handles operations like creating, joining, leaving, and updating information of a mashup (i.e. name and description).

In Listing 6.1, we see how the mashup logic is called by the respective API. Listing 6.3 shows an extract from the mashup logic module, i.e. the removal of a component. As we can see from this snippet of code, the function uses a custom method of the “Mashup” model which takes as arguments the mashup ID, the component ID, and two callbacks: success and error. The former is called if the operation succeeds; the latter in case a problem arises.

```

/**
 * Removes a component from the mashup
 * @param mashupId
4  * @param compId
 * @param {function(String)} success
 * @param {function({id, message})} error
 */
function removeComponent(mashupId, compId, success, error) {
9   Mashup.removeComponent(mashupId, compId
    , function(err) {
        if (err) error({ id: mashupId, message: err })
        else success(compId)
    }
14 )
}

```

Listing 6.3: Mashup Logic example, removing a component

A characteristic of Mongoose, in fact, is that it lets developers extend its objects, or more precisely the schema representation of the object (i.e. document), with custom routines to simplify database interactions. In Listing 6.4 we show a particular routine implementation, “removeComponent”, which is a static method added to the “Mashup” schema. Firstly, it finds the mashup using its ID; then, it removes the specified component model. As usual, given the asynchronous nature of these interactions, callbacks are used to control the flow.

```

mashupSchema.statics.removeComponent = function (mashupId, compId, cb) {
  this.findByIdAndUpdate(mashupId, { $pull: { components: compId } }
    , function(err, mashup) {
5      if (err) cb("Could not remove component from mashup")
      else {
          Component.remove({ _id: compId }, function(err) {
10             if (err) cb("Could not remove component")
             else cb(null, mashup)
          })
      }
  })
}

```

Listing 6.4: Database schema extension, removing a component

Finally, the mashup logic uses the component logic to create specific components. As described in Section 5.4, we decided to keep the concept of component as abstract as possible, so that components may be extended in the future by implementing new documents managed by the Mongoose module. In Jigsaw, we only implemented the default “PageComponent” which shows parts of a web page executed inside the headless browser.

6.3.2 Component Logic

The component logic handles all the operations that directly involve components, for instance moving and resizing components, and creating pages. Since the component is central to the infrastructure, pages are created inside components, and can only be shown inside components in the front-end. In a similar fashion to the mashup logic, this module uses the “Component” and “PageComponent” models implemented in Mongoose as described in Section 5.4. Listing 6.5 shows a snippet defining the move operation: it finds the specified component in the database model and changes its position. If the operation succeeds, it returns the component as JSON object, otherwise it returns an error.

```

3  /**
   * Moves a component
   * @param compId
   * @param {{ top, left }} newPos
   * @param {function(Component)} success
   * @param {function({id, message})} error
   */
8  function move(compId, newPos, success, error) {
    Component.findByIdAndUpdate(compId, { pos: newPos },
      , function(err, comp) {
        if (err || !comp) {
          error({ id: compId, message: "Could not move component" })
13        } else {
          success(comp.toObject())
        }
      })
  }

```

Listing 6.5: Component logic, moving a component

6.3.3 Page Logic

The page logic does not only manage the “Page” database model, but it also communicates with the browser logic to handle web pages that are open inside the headless browser. Furthermore, it handles the extraction of DOM views from web pages using a secondary module called “sourceExtractor” which is detailed at the end of this section.

The “Page” model, as described in Section 5.4, defines views and links. These are sub-documents managed by Mongoose, hence they cannot exist without the “Page” document itself: this means that each page has its own set of view and links. Listing 6.6 shows how views are created and stored, following the usual asynchronous model.

```

3  /**
   * Adds a new view
   * @param pageId
   * @param name view name
   * @param path the css path
   * @param {function(Object)} success returns the view
   * @param {function(Object)} error returns the error message
8  */
  function addView(pageId, name, path, success, error) {
    Page.findById(pageId, function(err, page) {
      var view = page.views.create({ name: name, path: path })

```

```

13     page.views.push(view)
    page.save(function(err) {
        if (err) error({ id: pageId, message: "Could not create view!"})
        else success(view)
    })
18 }

```

Listing 6.6: Page logic, adding a view

Furthermore, this module manages the browser logic: it can open and close a PhantomJS web page, as well as trigger any DOM event, for instance click, change and submit. These DOM events are initiated by the users in the front-end, sent to the server logic API, and finally to the page logic which communicates with the browser logic. However, before calling the respective functions implemented in the browser logic, this module checks the existing links between pages: matching links cause the DOM event to be replayed in the inter-connected web pages.

A link is defined as a triplet composed by event, tag, and element. The event may be any DOM event: in Jigsaw, however, we support only click, change and submit. The tag may be any string defining the link, for instance “myLinkedSearch”, which is sent by the client along the event and the target element. Finally, the element is the linked DOM element, for instance an input box, a button, or any other DOM element.

The process of inter-page communication through links operates as follows:

- find all pages containing the specified “(event, tag)” pair;
- for each page, retrieve the path to the DOM element; and
- trigger the DOM event in the specified element of the PhantomJS web page.

Listing 6.7 details this process implementation for the “change” event.

```

//find pages using the specified (event,tag) pair
2 Page.find({ 'links': { $elemMatch: { 'event': 'change', 'tag': tag } } },
    "_id links", function(err, pages) {
    pages.forEach(function(page) {
        page.links.some(function(link) {
            if (link.event == 'change' && link.tag == tag) {
                changeElement = link.element
7                 return true
            }
        })

        if (changeElement) {
12         browser.inputChange(page._id, { element: changeElement, value:
            change.value }
            , function() {
                success(changeElement)
                if (pageId != page._id) {
                    browserEvents.emitLinkedEvent("change", client,
17                     page._id, pageName, changeElement)
                }
            }
        )
    }
}

```

```

    }, function(e) { error({ id: pageId, message: e })
    })
    changeElement = null
  }
})
})

```

Listing 6.7: Page logic, replaying events between linked pages

Finally, the page logic may ask the browser logic for the source of a web page: if a particular view is specified, this function will return a modified DOM which only shows the DOM elements composing the view; otherwise it will return the whole DOM document. The source extraction process is detailed after introducing the browser logic itself.

6.3.4 Browser Logic

The browser logic bridges the whole server logic with the external PhantomJS process. The headless browser, in fact, runs on a different process and communicates with the server via a specific Node.js plug-in. This module is at the core of the remote control architecture: upon initialisation, it creates a PhantomJS instance and it opens the web pages stored in the database, if any. Inside PhantomJS, web pages are treated in the same way of tabs in a normal browser: they can be open, closed, and interacted with (e.g. clicking, refreshing, going back and forth through the history).

Once open, web pages react to a number of events which main purpose is to determine when a page is ready for being sent to the clients, i.e. when all resources finished loading. The most important event is “onLoadFinished”, which fires when the DOM of the web page is ready, and after each submit event (for instance a HTTP POST). At the end of “onLoadFinished”, we inject into the web page a subset of jQuery, which is used for easily retrieving DOM elements and, for instance, simulating clicks. This event, however, is not enough to guarantee that the client receive a full web page: certain pages, especially those that heavily rely on XHR/AJAX mechanisms, may request external resources after the DOM of the page is ready. Figure 6.7 shows the two different methods for determining when to send the web page.

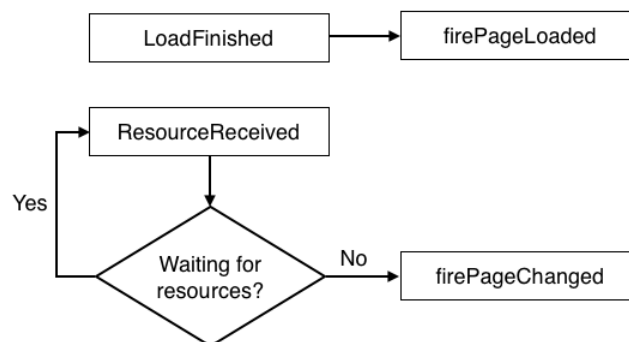


Figure 6.7: Implementation, web page flow

As can be seen from the snippet in Listing 6.8, implementing the “onResourceRequested” and “onResourceReceived” events, we increment a variable each time a resource is requested, and

decrement it each time a resource is received: when the count is zero, all resources should have been received and we can fire the page changed event. After some additional checks, the source of the page is sent to the page logic.

```

page.set('onLoadFinished', function () {
  page.injectJs('private/logic/jquery.js', function (status) {
    page.evaluate(function () {
      jQuery.noConflict()
5      })
      firePageLoaded(page)
      page.mash_state.requests = 0
    })
  })
10
page.set('onResourceRequested', function () {
  ++page.mash_state.requests
})
15
page.set('onResourceReceived', function (res) {
  if (res.stage == 'end') {
    --page.mash_state.requests
    clearTimeout(page.mash_state.reload)
    page.mash_state.reload = setTimeout(function () {
20      if (page.mash_state.requests == 0) {
        firePageChanged(page, "resource received")
      }
    }, 100)
  }
25 })

```

Listing 6.8: Browser logic, PhantomJS events

It is important to note that after the additional checks, and before being sent to the page logic, the current state of the HTML input elements present in the web page has to be hard-coded into the DOM. What the clients receives, in fact, is the serialised DOM of the page without the extra meta-data. In particular, we have to distinguish between HTML properties and attributes.

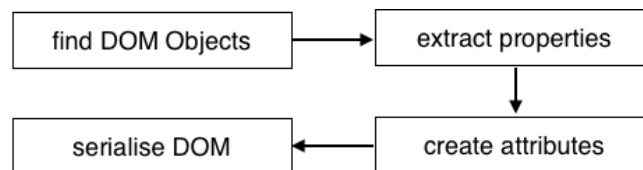


Figure 6.8: Implementation, DOM properties to attributes

DOM elements, like input controls, might show a value without having a “value” attribute: the value in this case is stored in a property inside the DOM object itself. When we serialise the DOM in order to send it to clients, properties are lost. Before doing so, therefore, we transform each property into its corresponding attribute as shown in Figure 6.8, thus hard-coding the value inside the HTML of the page. Using this approach, we do not need to handle extra meta-data manually in the client.

6.3.5 Source Extraction

After being sent to the page logic, as previously described, the serialised DOM has to be modified: firstly, for being correctly displayed in the clients; secondly, for state synchronisation reasons; finally, to create the various views that each different component is using. In order to understand the issues solved by this process, we need to introduce the mechanism used to show web pages in the front-end.

The clients receive a serialised DOM which may either represent the full web page, or views extracted from that web page. In order to transform the received DOM into an actual web page, we inject it into an iframe. The first issue, therefore, is obvious: we need to add the HTML “<base>” tag to the web page, so that instead of having to proxy the resources (which would greatly impact the server performance), the clients can fetch them from the third-party web sites themselves as shown in Figure 6.9.

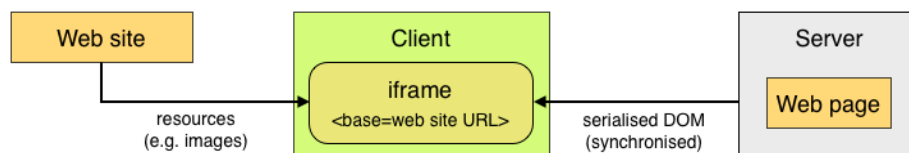


Figure 6.9: Implementation, serialised DOM injected into iframe

The second issue is related to JavaScript and it is slightly more complex. From the beginning, one of the main goals of this architecture has been defined as maintaining session state consistency across devices and web pages: since we allow the users to load any external web site into our infrastructure, we do not have any form of control over the JavaScript executed in these pages. This is not a problem for PhantomJS, the actual web page is in fact executed in a sand-boxed environment and it can, therefore, run any JavaScript without concerns.

It is a problem, however, for the copy of the web page, produced from the serialised DOM, which is injected into the iframe running on the client. The copy and the underlying web page on the server have to be synchronised at all times: while preventing default DOM events inside the iframe is not a problem; it is impossible to guarantee that JavaScript executing inside the iframe will not modify it in a different way than the JavaScript running inside the page in PhantomJS.

The only reliable way to solve this issue is to adopt a trade-off: we remove JavaScript completely from the front-end iframe, which is hosting the local version of the page (created from the serialised DOM), while leaving it in the original version of the page which executes in the back-end, as shown in Figure 6.10.

Not having JavaScript in the front-end iframe means that certain functionalities of web pages are limited. In particular, as we will discuss in the Technical Evaluation of Chapter 7, this can be noticed when trying to interact with those web pages that make heavy use of AJAX functionalities, for instance Facebook’s instant messaging system; or when complex drag and drop operations are required, for instance in Google Maps. However, this does not mean

that our infrastructure does not support JavaScript, in fact, most JavaScript operations can be executed in the back-end (inside PhantomJS) and the results sent to the client without particular issues.

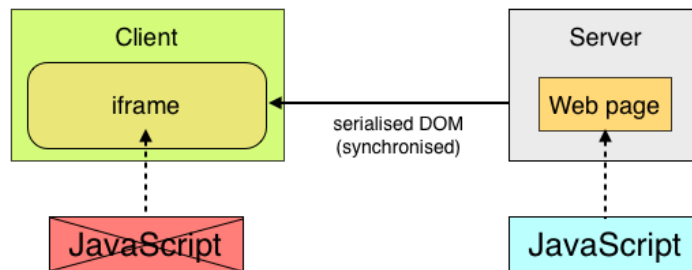


Figure 6.10: Implementation, server-side JavaScript execution

Finally, the source extraction module handles the creation of views by generating different versions of a web page’s serialised DOM. The entire procedure is shown in the following snippets of code. The page logic calls “getFullViews” which is detailed in Listing 6.9: this is the main function executed each time a web page has to be sent to the clients. As parameters, it takes the URL of the page, the source of the page extracted from PhantomJS (serialised DOM), an array of objects defining the views that have to be created, and a callback function called when the asynchronous process is done. This functions operates as follows:

- firstly, it loads the source of the web page into the Cheerio parser using the “loadSource” function detailed in Listing 6.10;
- then, it proceeds to creating the various views by calling the “extractSelection” function detailed in Listing 6.11;
- if a view is missing, i.e. if the DOM element defining the view cannot be found in the page, it creates a stub view that the client will interpret as an error;
- if views are not specified, it extracts the full body of the web page;
- finally, it executes the callback with the array of generated views which will then be sent to the client.

```

/**
 * Create full documents from the specified views, e.g.
 * <!DOCTYPE><html><head><head><body><body></html>
5  * @param baseUrl
 * @param source
 * @param {[{ viewId, path }]} views
 * @param {function(String, [{ viewId, source }]= )} cb
 */
10 function getFullViews(baseUrl, source, views, cb) {
    var viewSources = []
    var $doc = loadSource(baseUrl, source)
    var $originalBody = $doc("body").clone();
    views.forEach(function(view) {
  
```

```

15     var $viewBody = $originalBody.clone();
    if (view.path) { //create view
        var $selection = $viewBody.find(view.path)
        if ($selection.length == 0) { //not found
            viewSources.push({ id: view.viewId, source: "MISSING"})
        } else {
20             extractSelection($viewBody, view.path);
            $doc("body").replaceWith($viewBody)
            viewSources.push({ id: view.viewId, source: $doc.html()})
        }
    } else {
25         //no path, hence we take the full body
        viewSources.push({ id: view.viewId, source: $doc.html()})
    }
    })
    cb(null, viewSources)
30 }

```

Listing 6.9: Source extraction, main function

During the process of loading the source into the Cheerio parser, we start applying changes which affect the entire web page, as detailed in Listing 6.10:

- firstly, we strip any JavaScript from the source using a simple regular expression;
- then, we load the source into Cheerio and we proceed to remove certain DOM attributes which may try to use JavaScript (e.g. “onClick”);
- finally, we add the “<base>” tag to the “<head>” of the web page

```

/**
 * Strip JavaScript, load serialised DOM into parser, add <base> tag
 * @param baseUrl
 * @param source
5  */
function loadSource(baseUrl, source) {
    //strip script from page
    var SCRIPT_REGEX =
        /<script\b[^\<]*(?:!(?!\/script>)<[^\<]*>)*<\/script>/gi
    var noscripthtml = source.replace(SCRIPT_REGEX, "")
10    while (SCRIPT_REGEX.test(noscripthtml)) {
        noscripthtml = noscripthtml.replace(SCRIPT_REGEX, "")
    }

    //load serialised DOM source into Cheerio
15    var $doc = cheerio.load(noscripthtml);

    //remove the evil "onclick=" et al.
    $doc("[onclick]").removeAttr("onclick")
    $doc("[onchange]").removeAttr("onchange")
20    $doc("[onload]").removeAttr("onload")

    //add <base> to the head
    $doc("head").prepend('<base href="' + baseUrl + '>')
    return $doc
25 }

```

Listing 6.10: Source extraction, Cheerio parser

The returned “\$doc” object contains the modified source code parsed by Cheerio. From that objects, the “getFullViews” function extracts the body of the page, which is then given as parameter to the “extractSelection” function, detailed in Listing 6.11. As can be seen from the snippet of code, instead of completely removing the unneeded DOM elements from the current view, we only hide them. The reasons for this decision are twofold: firstly, removing DOM elements may cause undesired changes in the view we want to display, for instance breaking the CSS; secondly, we need to maintain the original CSS path to the elements contained in the view in order to reliably propagate DOM events (clicks, changes and submits). It would be possible to develop a function that completely removes the safe elements while emptying the needed ones (so that the relevant CSS paths would remain the same), however, at the moment of writing, Cheerio is not able to reliably handle such function.

```

5  /**
   * Extract selection (view) from body
   * @param $viewBody the full body
   * @param sel the CSS selector of the DOM tree
   */
function extractSelection($viewBody, sel) {
  var $element = $viewBody.find(sel);
  $element.siblings().attr("style", "display: none");
  $element.parents().siblings().attr("style", "display: none");
10 $element.attr("style", $element.attr("style") + "; top:0; left:0;
    margin:0; padding:0");
  $element.parents().attr("style", "top:0; left:0; margin:0; padding:0;
    background: rgba(255, 255, 255, 0); height: 100%; width: 100%");
}

```

Listing 6.11: Source extraction, view extraction

Moreover, after hiding the unwanted elements from the page, we move the view elements to the top-left corner: this helps displaying the view inside components in the front-end.

6.4 Client

The front-end, the features of which are detailed in Chapter 4, is a thin client. This means that while the cross-device mashup operations are executed in the back-end, the front-end is responsible for both initiating them and presenting their results. However, a certain amount of programming is needed in order to support collaborative features like notifications and highlighting, movable and resizeable components, and drag and drop operations. Furthermore, user interactions with both cross-device mashup elements and the web pages themselves (iframes positioned inside components), have to be captured and reliably reproduced in the server.

Leveraging the AngularJS framework, we divided the front-end into four modules as shown in Figure 6.11:

- the API module (described in Section 6.2.2) is responsible for sending and receiving socket messages;
- controllers interpret these messages creating a local data model, and are responsible for the behaviour behind the directives;

- directives define custom HTML elements specific to Jigsaw, for instance PageComponent, that the user can interact with;
- finally, the HTML created by the Jade compiler in the server, augmented with AngularJS syntax, is rendered by the browser.

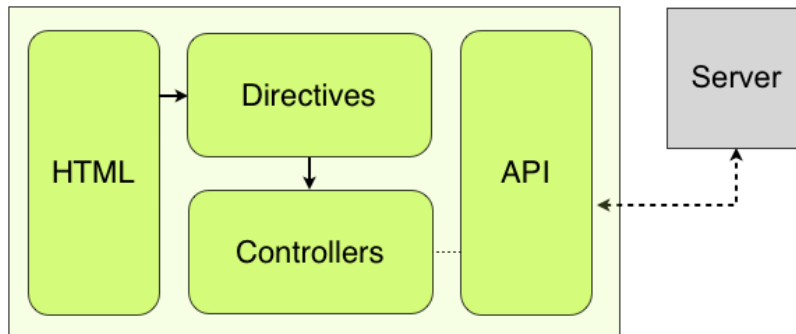


Figure 6.11: Implementation, front-end

6.4.1 Controllers

Controllers interpret API messages creating a local data model, and are responsible for the behaviour behind directives. We define a main controller that is responsible for general, application wide operations; and a controller for each section composing the front-end of Jigsaw: Home, Overview and Mashup.

Home controller

The Home controller handles the creation of new mashups, as well as the passive gallery of previews showing the existing mashups and their components, which can be used to join a particular mashup. Single operations like creating mashups, usually, are as simple as sending the proper message to the server and listening for a response. Listing 6.12 shows an example of such operations. The function for creating a mashup takes a name and a description (decided by the user in the view). Then, it sends the event “mashup:create” to the server, with the specified parameters and the screen resolution (useful for letting other users know how much space is available on the current device). Finally, it implements a callback that receives the created mashup which is loaded into the local data model.

```

$scope.createMashup = function(name, desc) {
  if (name && name != "") {
3    socket.emit("mashup:create", { name: name, desc: desc, size:
      Data.windowSize }
      , function(mashup) {
        $scope.mashups.unshift(mashup) //add as thumbnail
        $scope.data.mashup._id = mashup._id //set as current id
        $scope.loadMashup() //load it
8      })
  }
}

```

Listing 6.12: Example of operation initialisation, creating a mashup

AngularJS provides bindings between the local data model and the view. We use the “\$scope.mashups” array to describe the mashups shown in the preview gallery, therefore, as soon as we add the new mashup to the array, the gallery updates itself to show it. These bindings are automatic, and do not require any extra manual handling. For instance, the same is true for modifying the position of a component: as soon as the event with the new position is received, and the local data model is updated, the view automatically moves the component.

Creating new mashups is the only active operation the Home controller may initiate. However, in order to keep the gallery of previews updated, the Home controller has to listen for a number of events: from mashup creation, to component resizes and migrations. The previews, furthermore, do not show the actual web pages, but are colour-coded to distinguish between different web pages and views. As introduced in Section 4.1, the colour of the background represents the web page, while the colour of the border represents the view.

The process of colour-coding, depicted in Figure 6.12 works as follows: firstly, we take the ID of the element we want to colour-code; then, with a small library we create the MD5 hash of that ID, which is an hexadecimal string; finally, we extract the first six characters which are used as representation for red, green and blue in the standard hexadecimal colour format.

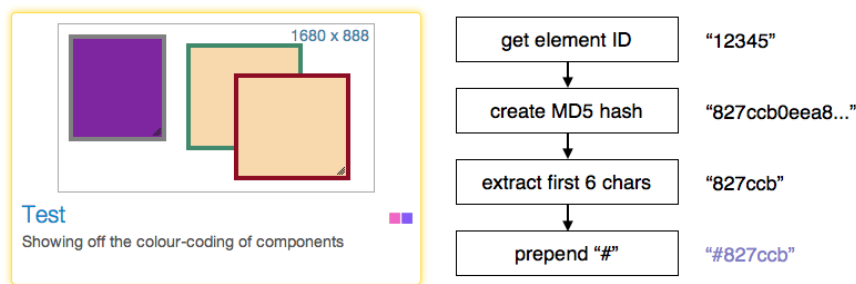


Figure 6.12: Colour-coding elements

Even in this case, AngularJS provides a quite useful mechanism: filters. Filters are small JavaScript functions which can be applied to any model variable directly from the HTML. Listing 6.13 shows the colour-coding filter for components, as well as its use in HTML.

```

//HTML
<div style="color: {{ comp.viewId | colourComponent:false }}"></div>
//JavaScript
filter('colourComponent', function() {
5   return function(text, newComp) {
        if (text)
            return "#" + md5(text).slice(0, 6);
        if (newComp)
            return "white"
10    return "gray"
    }
})

```

Listing 6.13: Example of AngularJS filters used for colour-coding components

Overview controller

The Overview controller handles the same events as the Home controller, except from mashup creation, and then more. In fact, the gallery in Overview may be used to move, resize, and migrate components from a mashup to the other: this means the Overview controller has to manage these operations, too. Instead of listing all the operations, which are the same we presented in Section 4.3, we only describe the move operation.

As shown in Listing 6.14, there is an inbound event as well as an outbound event. When the current user drags a component around, the “\$scope.componentMoved” is called, thus sending the “component:move” event with the new component position to the server. The server then will perform the move operation, updating the database model, and broadcast the result to all the interested users.

The “socket:component:move” event is fired upon receiving the message: firstly, it finds the correct mashup inside the “\$scope.mashups” array; then, the correct component in the respective “mashup.components” array; and finally it updates the component position. As previously mentioned, updating the data model is enough to fire a change in the HTML displaying the component.

```

//when current user moves a component
3 $scope.componentMoved = function(compId, pos) {
    socket.emit("component:move", { mashupId: this.mashup._id, compId:
        compId, pos: pos })
}

//on component moved by other users
8 $scope.$on('socket:component:move', function(ev, data) {
    var mashup = $scope.mashups.find({ _id: data.mashupId })
    var comp = null
    if (mashup) comp = mashup.components.find({ _id: data.compId })
    if (comp) comp.pos = data.pos
})

```

Listing 6.14: Example of front-end operation, moving a component

Mashup controller

The Mashup controller is the most complex of the implemented controllers. It handles not only the previously described operations, like moving, adding, and resizing components, but also events that interact with the web pages that have been injected into the iframes of the components. Given the complexity of this controller, we will only describe two operations: the click operation, which allows us to expand on the inter-communication of different web pages; and the component source event, which provides the web page source (serialised DOM) to the Component directive.

The click operation, initiated from the Component directive, is described in Listing 6.15. It takes as arguments a page ID, a target DOM element, and the click position. The first thing it does is to check whether the clicked DOM element is linked to any other element: if the link is found, it adds the link tag in the “page:click” message sent to the server; otherwise it simply sends a message with the page ID, the clicked element, and the click position.

```

//page actions (emit)
$scope.emitClick = function (pageId, target, pos) {
3   var link = this.comp.page.links.find({ event: "click", element: target
    })
    if (link) {
        socket.emit("page:click", { pageId: pageId, target: target, pos:
            pos, tag: link.tag })
    } else {
        socket.emit("page:click", { pageId: pageId, target: target, pos:
8         pos })
    }
}

```

Listing 6.15: Example of front-end operation, sending a click operation to the server

The server will then execute the click operation in the underlying web page running inside PhantomJS, and send back the updated page source to the interested clients.

The web page source, in form of serialised DOM, is received by the Mashup controller through the “socket:browser:fullSource” event described in Listing 6.16. This serialised DOM might refer to a full web page, or to a particular view. Furthermore, multiple components might be using the page that needs updating. In order to determine which components’ source has to be updated, we look into all components that are using the specified web page and the specified view, if any. When a pertinent component is found, we send the serialised DOM to the Component directive using a local broadcast.

```

1 //component source
$scope.$on('socket:browser:fullSource', function (ev, data) {
    var pageId = data.pageViewId.split("_")[0]
    var viewId = data.pageViewId.split("_")[1]
    //update components using this page
6    $scope.mashup.components.forEach(function(comp) {
        //if this is the right page
        if (comp.page && pageId == comp.page._id) {
            //if the component is using the right view
            if ((comp.view && comp.view == viewId) || (!comp.view &&
11             viewId == "full")) {
                $scope.$broadcast("source"+comp._id, data.doc)
            }
        }
    })
})

```

Listing 6.16: Example of front-end operation, receiving serialised DOM

6.4.2 Directives

The aforementioned Component directive is the most important directive in Jigsaw: the entire front-end revolves around components. AngularJS directives, in general, define custom HTML elements specific to the application, which the user can interact with as normal HTML controls. This is where the HTML elements shown in Jigsaw are created.

We provide the following list with a description of the most significant directives. With exception for the first one, all these directives are used in the Mashup page as shown in Section 4.2:

Thumbnail Two distinct directives are used for creating thumbnails that are shown in the preview galleries of Home and Overview: the former only shows the mashup with its colour-coded components, while the latter additionally allows moving and resizing the components.

MashupArea This directive lets users create components by highlighting a rectangle in the mashup background with a dragging operation.

NewComponent A newly created component does not host any web page: this directive represents such components, allowing for the creation or selection of a web page.

Component The directive representing a full component encapsulating a web page, which supports the flip action showing the configuration panel.

LinkSelection This directive provides the link selection operation.

ViewSelection This directive provides the view selection operation.

Instead of describing in more details each directive, we will focus on the Component directive partially shown in Listing 6.17.

```

//Page Component directive
directive('Component', ['$timeout', function (init) {
    var linkFn = function(scope, element, attrs) {
        //wait for directive DOM ready
5        init(function() {
            //function for flipping component revealing configuration mode
            //function for initiating view selection
            //function for initiating link selection
            //function for jQueryUI drag and drop operations (resize, move)
10         //source received event (local broadcast)
            scope.$on("source"+scope.comp._id, function(ev,source) {
                scope.comp.isLoading = false
                $iframe[0].contentWindow.dist_contents = source
                $iframe[0].src = 'javascript:window["dist_contents"]'
15            })
            //capturing iframe events (click, change, submit)
            $iframe.load(function () {
                $iframe.contents().off("click")
                .on("click", function (event) {
20                    event.preventDefault()
                    event.stopPropagation()
                    event.stopImmediatePropagation()
                    var pos = { left: event.pageX, top: event.pageY }
                    var cssPath = jQuery(event.target).getPath()
25                    scope.emitClick(pageId, cssPath, pos)
                    return false;
                })
            })
        })
    })
30    //listeners for other events

```

```

    // - URL change
    // - collaboration highlighting
    // - ...
  }
35  return {
    restrict: 'E'
    , templateUrl: '/t/pageComponent'
    , replace: true
    , scope: true
40    , link : linkFn
  }
  })

```

Listing 6.17: Component directive

Directives are built as JavaScript closures: each directive has a name (in this case “Component”), a number of arguments (here we only use “\$timeout” to wait for DOM ready), and a function returning the object representing the closure. This object may have a number of keys, but three are required and must be present: `restrict`, `template` (or `templateUrl`), and `link`.

`Restrict` defines how a directive may be used in the HTML. For instance, “E” means “element” and “A” means “attribute”. We decided to use this directive as a normal HTML element, therefore in HTML we can call it by writing “<component></component>”.

`TemplateUrl` refers to the HTML file defining the graphical appearance of the directive: in this case, we simplified our “pageComponent” template in Listing 6.18. A component is a flip panel with an `iframe` in the front and the configuration options in the back. At the bottom it has a bar showing various buttons as presented in Section 4.2. The flip animation, which shows either the execution mode (front) or configuration mode (back) is a simple effect written in CSS3.

```

<div>
  <div class="flip-container">
3    <div class="flipper">
      <div class="front">
        <iframe src="about:blank"></iframe>
      </div>
      <div class="back">
8        <div class="configuration"></div>
      </div>
    </div>
  </div>
  <div class="bottom-bar"></div>
13 </div>

```

Listing 6.18: Component directive, HTML template

`Link` is the linking function which determines the behaviour of the directive: JavaScript modifying the directive’s template, or interacting with the Controller is defined inside this function. In the snippet of code reported in Listing 6.17 we only show a small portion of this function. In particular, we show the listener for the aforementioned broadcasting of web page source from the Controller; and the method we use for intercepting DOM events, for instance clicks, resulting from user’s interaction with the web page.

Upon receiving the local broadcast containing the serialised DOM of the web page that the component needs to update, we have to inject it into the component's iframe. As it turns out, there are various ways to do this²⁵. After some testing, we discovered that the best method in terms of speed and compatibility with browsers, is to make use of the JavaScript URI scheme as shown in Listing 6.17, line 10.

In order to intercept DOM events, we use mechanisms for adding element listeners provided by jQuery. However, capturing the event and preventing its natural course, is not enough: we needed a way to tell the server what element exactly has to be clicked inside PhantomJS without having to rely exclusively on IDs, which might not be always present in the clicked elements.

To do so, we extended jQuery with the “getPath()” function detailed in Listing 6.19, which provides the CSS path to the target element in a smart way: it goes up the DOM tree from the element until it either finds an ID, or it arrives to the top. Once done, it returns a string representing a valid CSS selector pointing to the clicked element.

```

jQuery.fn.getPath = function() {
2   var el = this[0] //standard DOM element
    var names = [] //CSS Path tree
    while (el.parentNode) {
        var $el = jQuery(el)
        var elName = el.nodeName.toLowerCase()
7       //html and body selectors are not needed
        if (elName == "html" || elName == "body") {
            el=el.parentNode
            continue
        }
12      //as soon as an element in the tree has an ID, we can break and
        return
        if (el.id) {
            names.unshift(elName+'#'+el.id)
            break
        }
17      //otherwise, we create the tree
        else {
            //single element
            if (el == el.ownerDocument.documentElement) {
22                names.unshift(elName)
            }
            //no siblings of same kind
            else if ($el.siblings(el.nodeName.toLowerCase()).length == 0) {
                names.unshift(elName)
            }
27          //has siblings of same kind
            //use :nth-child instead of :eq which is not supported by
            Cheerio
            else {
                //count the siblings before el
                var prevSiblingsCount = $el.prevAll().length + 1
32                names.unshift(elName+":nth-child("+prevSiblingsCount+")")
            }
        }
    }
}

```

²⁵<http://sparecycles.wp.com/2012/03/08/inject-content-into-a-new-iframe>

```

    //next element in the path
    el=el.parentNode;
  }
37 }
  //return the CSS path selector
  return names.join(">")
}

```

Listing 6.19: CSS Path creation

6.4.3 HTML

To conclude the description of Jigsaw's implementation, we briefly show the difference between writing HTML and using the Jade templating language. As introduced in Section 6.1, web pages written in Jade are compiled server-side before being sent to the client: in Jigsaw, every template is written in Jade.

For instance, in Listing 6.20 we show again the simplified HTML structuring a Component directive.

```

<div>
  <div class="flip-container">
    <div class="flipper">
      <div class="front">
5        <iframe src="about:blank"></iframe>
      </div>
      <div class="back">
        <div class="configuration"></div>
10      </div>
    </div>
  </div>
  <div class="bottom-bar"></div>
</div>

```

Listing 6.20: Component directive, HTML template

Listing 6.21 shows the actual Jade that would produce the previous HTML snippet, as it would have been written in Jigsaw.

```

div
2  .flip-container
    .flipper
      .front
        iframe(src='about:blank')
      .back
7      .configuration
    .bottom-bar

```

Listing 6.21: Component directive, Jade template

We believe writing Jade templates instead of standard HTML greatly increases productivity, as well as guarantee compatibility with the current and future HTML standards.

7

Technical Evaluation

Jigsaw has been developed with the goal of allowing the creation, management and use of cross-device mashups, through direct manipulation of existing web applications, without requiring a specific browser, or the knowledge of any programming language.

The purpose of the technical evaluation presented in this chapter is to study the versatility and compatibility (breadth) of the Jigsaw infrastructure with a variety of web sites, as well as highlighting technical challenges and trade-offs.

The study, which takes inspiration from an evaluation model proposed in Nebeling et al. [20], has been carried out by reproducing a standard cross-device mashup scenario on 50 top web sites as ranked, according to popularity and traffic, by Alexa¹. In particular, we tested the first 5 web sites from 10 categories: Arts, Business, Games, Health, Home, News, Science, Shopping, Society, Sports.

On each iteration of the scenario, i.e. for each web site, we assessed a number of criteria by creating the entities (mashups, components, views) as well as performing the principal operations (mashing, distributing, and (co)browsing) presented in Chapter 3.

7.1 Criteria

This technical evaluation focuses on studying the compatibility with a variety of target web sites which a user might desire to use as part of the cross-device mashup. In order to evaluate Jigsaw, then, we follow the aforementioned operations required by these environments.

Therefore, we propose three groups of criteria: Browsing, Distributing, and Mashing.

¹<http://www.alexa.com>

Browsing

A web site loaded into a Jigsaw component should behave as if it were in the native browser, therefore we assess the following general browsing criteria:

- page elements are loaded and displayed correctly
- page elements behave as expected
- user interactions are handled correctly

Distributing

It should be possible to create, distribute, and use components, therefore we assess the following distribution criteria:

- page elements can be extracted and distributed as components
- distributed elements are displayed correctly
- distributed elements behave as expected
- distributed elements are synchronised with the underlying web page

Mashing

Components should allow setup and use of inter-page communication, therefore we assess the following mashup criteria:

- elements of a web page can be linked with elements belonging to another page
- click, change and submit events are correctly replayed in linked elements

Supporting multi-modal user interactions falls outside the scope of this thesis. Hence, touch interactions, for instance, are only partially supported at this time. Therefore, these will not be taken into account during this evaluation. Furthermore, we initially intended to test an additional set of criteria relative to the various screen resolutions and device sizes. However, since highly dependent on the particular cross-device mashup evaluated, we determined such criteria inconclusive. Therefore, we decided to not include them in this technical evaluation. We will nonetheless describe a related issue at the end of this chapter, i.e. the resolution of web sites open into the headless browser is fixed and shared by all connected devices.

Rating

Each criterion has been independently measured with the following 5-point scale rating:

1. very poor, major issues (e.g. page will not load)
2. poor, only specific elements are working
3. fair, most critical elements are working
4. good, only specific elements are not working
5. very good, full support

However, it is important to note that a web site which is rated low in “Browsing” will affect the “Distributing” and “Mashing” criteria evaluation, i.e. compatibility-wise, the “Browsing” criteria takes precedence over the other two. For instance, it might happen that components can be successfully extracted and distributed, while the elements inside the components are broken.

7.2 Scenario

In order to rate the various criteria assessed in this evaluation, we propose a simple but representative cross-device mashup scenario, which can be constructed using almost any typology of web sites. The scenario is composed of two devices, each using a different mashup, and two different inter-connected web sites. These are separated into components, which are then distributed and tested. In particular, we identify three categories of components: content, showing only the main content; menu, showing only the menu; and search, showing only the search bar of the web site.

Furthermore, only one of the two web sites is changed at each iteration of the scenario, while the other is fixed. The purpose of the fixed web site is exclusively to test the mashing capabilities (i.e. inter-page communication) of the cross-device mashup. This web site should not impact the evaluation, therefore we use one which, if evaluated, would receive an excellent rating. The selected web site is Wikipedia: it can be easily separated into components, and it has been determined that it does not produce any problem in Jigsaw.

Figure 7.1 shows an example of such scenario using Wikipedia and Reddit. Each mashup is accessed by a different client and shows various components:

- the one on the left has the full Wikipedia web page, the extracted search bar, and two components showing different parts of the Reddit web site;
- the one on the right has the full Reddit web site, the extracted search bar and both the menu and main content of Wikipedia.

The red, dashed arrow symbolises the inter-page communication which has been established linking the two respective search bars, while the black arrows symbolise the extraction and distribution of components.

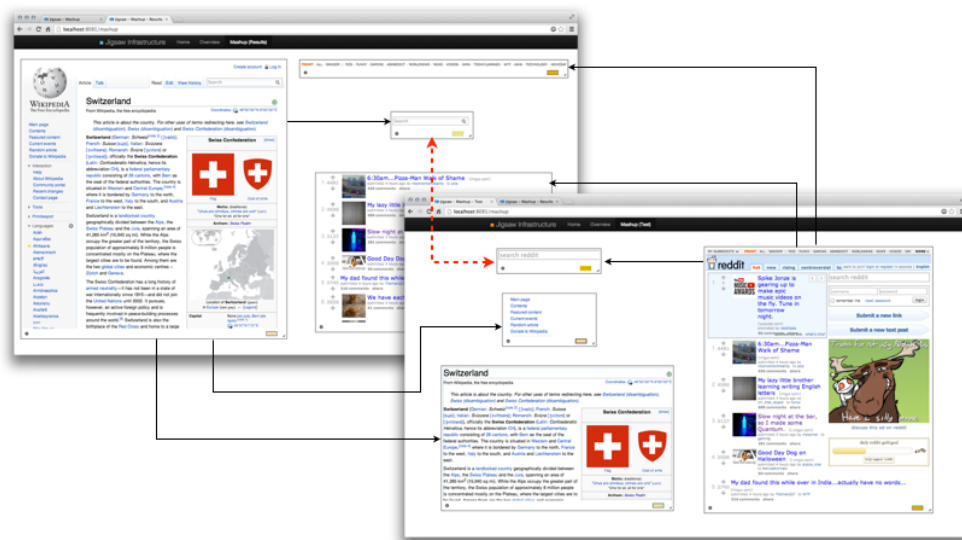


Figure 7.1: Technical evaluation, example scenario

The full evaluation procedure, after having created the fixed components, is the following:

1. create a new component loading the target web page from the list;
2. evaluate general “Browsing” criteria;
3. select, extract and distribute the three components;
4. evaluate “Distributing” criteria;
5. link pages together by selecting the respective search forms;
6. evaluate “Mashing” criteria;
7. clear the mashup and restart with the next web site.

7.3 Results

In this section, we will firstly present and comment the assessed criteria by showing the related graphs; then, we will provide a summary of the global ratings; finally, we will provide a summary of the most relevant issues discovered. The full data produced during this evaluation, which might be useful for comparison with other approaches in the future, can be found in Appendix B.

Browsing

Figure 7.2 shows the results collected for the Browsing criteria, which focus on the appearance, behaviour and support of user interactions provided by the web site injected into the components.

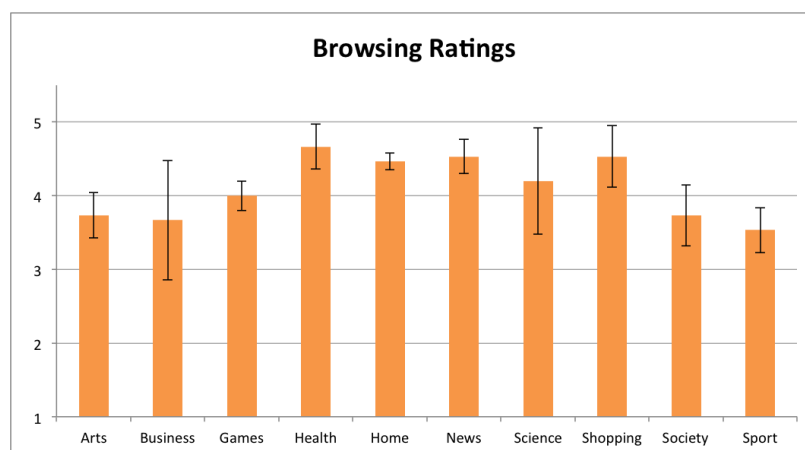


Figure 7.2: Technical evaluation, results of Browsing criteria

The ratings in average are good with a mean of 4.1 and a standard deviation of 0.4. This means that in general, most of the tested elements were working as expected. In particular, Health, Home, and News web sites worked best, with a mean around 4.5. Sport, which is the

lowest rated category, has a mean of 3.5 with a standard deviation of 0.3: this means that in average, even in the lowest performing category the most critical elements worked.

Even though in average the Browsing result are good, from this first part of the evaluation we could already determine most of the issues affecting Jigsaw. It is not surprising that Sport scored the lowest: these web sites have bloated graphics-heavy designs, Flash animations, continuous background AJAX operations, and multiple internal iframes. On the other hand, it is also not surprising that accessibility driven web sites, like those in the Health category, worked better. We noticed that in general JavaScript is hard to handle, in particular for web sites like Google Translate, Twitter, and Facebook which heavily rely on JavaScript for expanding input boxes on click, managing AJAX side-loaded components, providing auto-completion, and other functionalities. Some of these issues will be summarised at the end of the chapter.

Furthermore, since we are using a remote control architecture where web pages are loaded inside a headless browser (PhantomJS) running in the back-end, serialised DOM has to be sent to all clients each time a DOM change occurs. PhantomJS, however, cannot identify DOM changes. Therefore, we use the resource requested/resource received method described in Section 6.3, which allows us to send the whole web page to the clients. The implications of this trade-off are twofold: firstly, there is no easy and reliable way to determine what DOM element actually changed, thus we need to send the whole serialised DOM each time a resource is received; secondly, DOM changes that do not require external resources cannot be detected (for instance a ticking JavaScript clock updating a div).

Distributing

Figure 7.3 shows the results collected for the Distributing criteria, which focus on creating views by extracting elements from the web pages, as well as on the appearance and behaviour of distributed components.

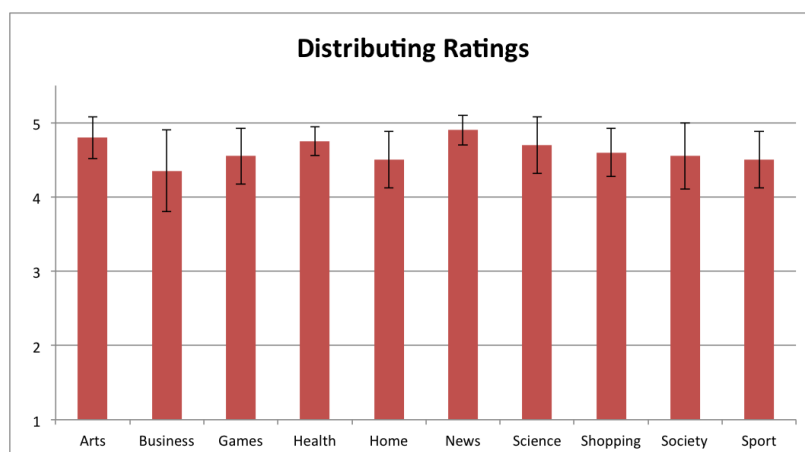


Figure 7.3: Technical evaluation, results of Distributing criteria

In average, the ratings are between good and very good with a global mean of 4.62 and a little standard deviation of 0.17, with News (mean 4.8, standard deviation 0.2) and Health (mean 4.75, standard deviation 0.2) categories again among the best. The lowest rated is Business, with a mean of 4.35 (more than good) and a standard deviation of 0.5.

Evaluating this set of criteria, we encountered few small issues. In some cases, we found small CSS inconsistencies between the original element and the extracted one, for instance it might happen that the extracted element is not positioned correctly inside the component, or that the page background shows, out of place, as background of the element. In other cases, we had problems when trying to extract iframes from the web pages. Finally, in even rarer cases the Cheerio library that we use for extracting DOM did not handle elements correctly (bug).

Mashing

Finally, Figure 7.4 shows the results collected for the Mashing criteria, which focus on inter-page communication, and, therefore, creating links between elements belonging to different web pages as well as replaying events on these elements.

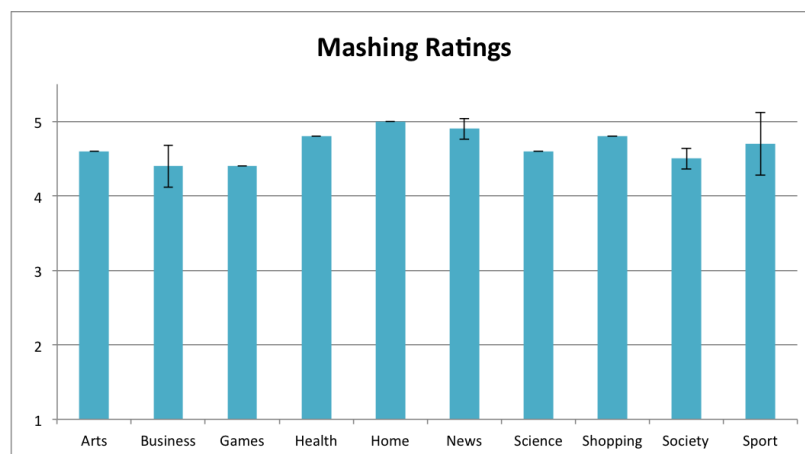


Figure 7.4: Technical evaluation, results of Mashing criteria

Similarly to the Distributing criteria, in average the rating are between good and very good, with a mean of 4.67 and a standard deviation of 0.2. In regard to the Mashing criteria, the best rating category is Home with a perfect score of 5, followed by News (mean 4.9, standard deviation 0.14). Among the lowest scoring categories we find Business with a mean of 4.4 and a standard deviation of 0.2, which is however a good rating.

During this part of the evaluation, we found that certain web sites evolve their DOM, for instance by randomly changing forms ids to prevent spam, thus breaking links between elements. Apart from this problem, we found only few, non-critical issues. However, we should note that the linking of web pages only determines connections between user triggered events, i.e. with this implementation we cannot program two web sites to remotely communicate in the back-end by observing each other. Our linking mechanism is, therefore, quite simple.

Furthermore, it is important to note that although both the Distributing and the Mashing criteria are rated highly, in the most cases we do not take into account problems that, influencing the Browsing ratings, may cause web sites to work poorly. The reasoning behind this decision is that even though some underlying functionality of the web site element might be missing, the actual operation which is evaluated (e.g. creating a component) is successful.

Overall

Overall, as summarised in Figure 7.5, Jigsaw offers good compatibility with most web sites, and both Distribution and Mashing rate from good to very good. Moreover, a number of issues affecting the rating negatively arise from limitations of the headless browser itself (PhantomJS), which might be greatly improved in the next months with the new release of version 2.0².

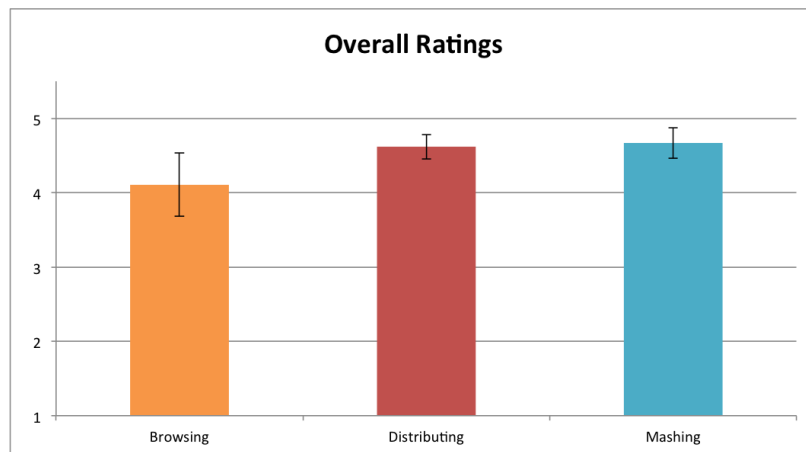


Figure 7.5: Technical evaluation, overall ratings

We end this chapter by summarising some of the most important open issues discovered during the technical evaluation:

- Related to the headless browser itself, we found that not supporting MutationObserver³ it cannot detect changes in the DOM and therefore we can only update the clients when resources are received. Furthermore, videos (e.g. in YouTube) cannot be displayed because not supported by PhantomJS at this point. Both issues, however, will be solved with the release of version 2.0.
- Since web pages are loaded inside the headless browser, they are set to a default resolution which might not coincide with the resolution of the client device using the web site. This problem is visible, for instance, with the auto-complete mechanism of Google Search, which centres the suggestion box relative to the headless browser resolution, thus hiding it from small components using that page.

²<https://github.com/ariya/phantomjs/issues?milestone=12>

³<https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver>

- We have no control over iframes contained inside the loaded web pages in the back-end, therefore we cannot reliably synchronise them by replaying events. Furthermore, we cannot prevent JavaScript injections coming from these iframes, i.e. the local copy of the page might act differently than the original in the back-end.
- In order to prevent de-synchronisations and duplication of updates to the external web servers, we removed JavaScript from the local copy of the page in the front-end. This does not affect the normal behaviour of the page in most cases (since events are carried out in the back-end), however, it limits some special cases where complex AJAX operation are expected to be available directly in the client. For instance, we cannot reliably implement drag and drop of the Google Maps page, or use the Facebook chat.
- Finally, Jigsaw, like other similar re-authoring systems, cannot handle evolving web sites like those that randomly change the ids of their DOM elements, or even entire parts of the DOM structure.

8

Conceptual Evaluation

This chapter presents a conceptual evaluation of the Jigsaw infrastructure. We will firstly introduce the logical framework used for the evaluation, and we will then assess its various dimensions by applying it to Jigsaw in form of a discussion.

8.1 Framework

The presented conceptual evaluation is largely based on an existing framework describing various design dimensions that can help in better understanding the features provided by tools and applications for multi-device environments [22]. We do, however, extend this framework with dimensions relative to mashups and collaborative browsing, which have been identified by analysing related work in the area and by building our infrastructure for cross-device mashups.

8.1.1 Multi-Device Environments

In terms of general multi-device environments, we report the following dimensions presented in Paternò and Santoro [22]:

Distribution This dimension analyses the distribution of the user interface elements across various devices at a given time. It can either be *static*, if the configuration of such distribution is fixed; or *dynamic*, if the the distributed UI elements can be duplicated and moved across devices at run-time.

Migration This aspect analyses whether the infrastructure supports continuity, i.e. the ability of saving the current session and continuing working on it after having migrated the elements to a different device. The value of this dimension is the *set of elements* supporting migration. In this particular case, we refer to web applications and therefore consider for instance state of forms, JS variables, and cookies.

Granularity UI distribution and migration might be defined at various levels of granularity: the *entire UI*; only *groups of UI elements*; any *single UI element*; or the *interactive UI elements*.

Trigger Activation Type This dimension outlines how migration and distribution of UI elements is performed: it can either be *manual*, when the users have to interact with the elements, for instance with push and pull mechanics; *system*, when the infrastructure can automatically modify the distribution and migration of components; and *mixed*, when both trigger types are available.

Device Sharing between Multiple Users Here, the considered aspect is the way multiple users can share devices. This can be done by *moving* information to the shared device, or by *interacting* with the shared device.

Timing The evaluated infrastructure might require devices to *immediately* handle migrated elements, allow *deferred* migration, or both (*mixed*). In a deferred migration, a device might move an element to a second device even if it is not available (i.e. once available the device should be able to receive the element and maintain continuity).

Interaction Modalities Involved This aspect analyses the interaction modalities supported by the infrastructure (e.g. touch interactions), in particular it distinguishes between *mono-modal*, when only one type of interaction is supported; *trans-modal*, when multiple types of interaction are supported but only one at the time (e.g. either mouse or touch); and *multi-modal*, when devices support more than one type of interaction at the same time (e.g. mouse and touch).

Generation Phase This dimension specifies whether the distributed elements are created at *design-time*, at *run-time*, or in both phases (*mixed*). Pre-built components, for instance, are generated at design-time; while components made of elements extracted from web pages are usually created at run-time.

Adaptation Aspects When changing the currently used devices, UI adaptation might be required. This can be achieved by *scaling*; *transducing*, i.e. changing the format, for instance by compressing images; or *transforming* elements, i.e. modifying both contents and structure.

Architecture In general, a multi-device environment might be built with either a *peer-to-peer* architecture; or a *client/server* architecture.

8.1.2 Cross-Device Mashups

Furthermore, for evaluating cross-device mashups in particular, we propose the following additional dimensions identified by analysing related work as well as by building our infrastructure:

Synchronisation Modality Distributed UI elements should be able to communicate between each other, i.e. it should be possible to synchronise them. This might be *automatic*, if the elements share the same data source; *manual*, if the connection has to be manually established; and *mixed* if both cases are possible.

Synchronisation Scope Furthermore, the synchronisation might be *component based*, i.e. at the level of the element container; *page based*, i.e. at the level of the underlying web page; or *element based* in case elements are directly connected.

Synchronisation Consistency This aspect defines how complex it is to maintain synchronisation across devices and web sites. In a *consistent* system, resynchronisation is easy to achieve; while in a *inconsistent* system, resynchronisation is hard to achieve. An example of similar observations might be found in Lowet and Goergen [17] (robustness of JavaScript engine output synchronization).

Type of Components This dimension analyses the type of components used in the distribution. These can be *widgets*, i.e. small, pre-built, self-contained, web applications; or *web components*, intended as parts of web sites.

Components Creation Here, we define the mechanisms that can be supported for creating components. Components might be *pre-built*, i.e. widgets; *scripted* at design-time, i.e. by developers working on the cross-device-mashup; or generated by *direct-manipulation* of web pages.

UI Behaviour Depending on the architectural decisions, DOM elements exposed inside the front-end components might behave as if in a native browser window, or in a less natural way. The behaviour, therefore, might be defined as *expected*, or *unexpected*. An example of similar observations might be found in Lowet and Goergen [17] (UI experience).

Robustness to Errors As observed in Highlight [21], re-authoring web page content might introduce errors occurring inside the components: these problems might affect the entire cross-device mashup (*low robustness*), or be limited to the particular web page (*high robustness*).

Flexibility to Changes This dimension analyses the flexibility to changes, e.g. because of evolving web sites, in the source of the re-authored pages [1]. It might be *high* if components can adapt well to the new source, or *low* if no adaptation is provided.

Type of Collaboration The infrastructure might support different types of collaboration: *co-located*, if the users work in geographical proximity; *remote*, if the users do not work in proximity; or *both*.

Scalability This dimension defines whether the cross-device mashup infrastructure is *scalable*, i.e. both front-end and network traffic support a large number of users; or *not scalable*.

8.2 Discussion

Architecture

Jigsaw is based on a client/server remote control *architecture*, where the whole cross-device mashup lies on the server. The thin client, therefore, only initiates operations which are then carried on in the back-end: this is true for structural operations, like moving a component,

but also for user interactions with the web pages used by the mashups (e.g. filling a form). These interactions are captured and sent to the server, which executes them inside a headless browser. Results are then sent back to the client, which updates its local copy of the page. This architecture allows for *consistent state synchronisation* across web sites and devices, but at the expenses of *scalability* and, more importantly. Furthermore, the *UI behaviour* might be unexpected in certain cases because of the remote control mechanism itself, as shown in Chapter 7.

Collaborative Browsing

Scalability, however, is not an issue in Jigsaw: this infrastructure targets mainly *co-located collaboration*. Therefore, users might work in environments like the office of the future or the interactive living room, where it does not make sense to have more than a dozen clients sharing mashups. Furthermore, multiple users might *share devices* both by moving information to one (e.g. large screen), and by collaboratively work on the same device (e.g. interactive tabletops).

Components

Information is represented as web sites, or *parts of web sites*, which are encapsulated into distributed components. These components are not fixed: they are *generated at run-time* by *direct-manipulation* of web pages. This means that any part of a web site might be encapsulated into a component, without the need for complex modelling or pre-built widgets. The UI *granularity* therefore goes as deep as the single HTML element, and as far as an entire web page. However, a component might only contain a single DOM tree, i.e. we do not provide solutions for selecting and displaying multiple separate elements in the same view.

Operations

Components might be dynamically *distributed at run-time*. The infrastructure, in fact, supports all the distribution operations described in Section 3.1.2: components may be moved, cloned (i.e. duplicated maintaining the same underlying web page), copied (i.e. duplicated creating a new instance of the underlying web page), resized, deleted and migrated from a mashup to the other. Users initiate these operations in the front-end, thus the *trigger activation* type is manual. However, it is important to note that at this point Jigsaw is only *partially multi-modal*, i.e. it fully supports mouse interactions, while touch interactions can be used with existing cross-device mashups, but not for creating one.

Synchronisation

As previously introduced, state synchronisation is provided by the remote control architecture itself. *Continuity* during device migration, therefore, is implicitly guaranteed for all elements, i.e. devices joining a mashup automatically inherit the current state of that mashup and all its components, with respective web pages. Moreover, since mashups exist on the server and are independent from the connected devices, the *timings* for both migration and distribution of components are not an issue. For instance, a device might modify a mashup which has been used by another device that at the moment is offline, when the latter

reconnects, all changes to the mashup are automatically re-synchronised. In other words, in terms of synchronisation between devices and mashups, the system is *consistent*.

Between components, however, synchronisation is harder to guarantee: Jigsaw uses both *automatic and manual component synchronisation*. Automatic synchronisation is *page based*, which means that two components sharing the same underlying web page will always be synchronised. Manual synchronisation, on the other hand, is *element base*, which means that in order for two different pages to be synchronised, the user has to manually link elements together. This kind of inter-page communication can only handle replay of user triggered events, i.e. users cannot program two web sites to remotely communicate in the back-end by observing each other.

Web Sources

The web pages used by Jigsaw are external to the infrastructure: any web page can be used to construct components. This means that it has to be *robust to eventual errors* introduced by those pages that are outside its direct control. Because of the headless browser, this is not a problem: even if a page crashed, the other pages involved in the cross-device mashup would not be affected, and the crashed page itself could be reloaded from the front-end. Another problem introduced by handling external web pages, is that their structure could change at any time (i.e. they might evolve). Components, therefore, have to be flexible enough to guarantee functionality even if the current structure of the encapsulated web page changes, i.e. when the current view showed by the component cannot be found. In Jigsaw we avoid this problem by *switching to the full page view*, and by giving the task of recreating eventual broken views to the user.

As a final remark, we should note that the only *UI adaptation* provided in this project is resizing components. However, we do not scale nor transform the content of the component. Therefore, small screen devices might be problematic to use.

9

Conclusions

In this thesis we presented Jigsaw—an infrastructure for cross-device mashups. To this end, we elaborated the concept of cross-device mashups, introduced in MultiMasher [14], by exploring related research areas of mashups, distributed user interfaces, and computer-supported cooperative work (in particular collaborative browsing), as well as by developing two use case scenarios.

The basic idea is to enable multiple devices to access multiple web sources in a coordinated manner. Cross-device mashups, therefore, are collections of mashups used by multiple cooperating devices. In particular, each mashup is composed of a number of web sites combined together in the form of distributed components, each showing a different part of a web page. Platforms for cross-device mashups should support features of distributed user interfaces and mashups, while providing solutions to the challenges arising from their combination and the co-browsing environment. For instance, components might be duplicated or moved across mashups, while mashups might be migrated between devices: users should not only be able to perform such operations, but also be aware of each other's. Furthermore, synchronisation issues should be appropriately addressed, thus guaranteeing continuity in the form of migration from one device to another.

The Jigsaw infrastructure supports creation, management and use of cross-device mashups by focusing on direct manipulation of existing web applications, without requiring particular tools or programming knowledge. Its users, therefore, are able to perform the aforementioned operations by using a normal browser, through drag and drop interactions. Continuity and synchronisation issues are addressed by employing a centralised remote control architecture. Consequently, both distribution and mashup operations, as well as interactions with web pages encapsulated into components, are initiated by the clients but carried out in the server. User interactions with web pages, in particular, are not executed inside the local browser, but in a headless browser running on the server: results are then handled and propagated to the connected devices accordingly.

Jigsaw has been studied with both a technical and a conceptual evaluation. The goals of the technical evaluation were twofold: on one hand, we wanted to test versatility and compatibility of this infrastructure with a variety of top web sites selected according to popularity and traffic; on the other hand, we wanted to highlight technical challenges and trade-offs adopted. Results were positive: in general Jigsaw handled web sites from good to very good, with a global average rating of 4.5 over 5. The goal of the conceptual evaluation was to assess the infrastructure with a number of dimensions defined by a logical framework: this could be useful, for instance, for comparing the infrastructure with future related projects.

Finally, Jigsaw has been successfully used for developing the applications described in the use case scenarios, it was presented during the study information days at ETH Zürich without observing particular problems, and it is currently showcased and demoed to visitors of the GlobIS research group.

Future Work

In future work, looking at cross-device mashups in terms of a mashup-driven approach to DUIs, researchers could investigate the use of Jigsaw as a general tool for developing cross-device user interfaces.

The Jigsaw infrastructure itself could be extended by solving the major open challenges identified during the technical evaluation, in particular:

- The current headless browser cannot identify DOM changes. This means that while we can detect HTTP requests involving external resources, local changes inside the page (for instance a visual count-down modifying a div) do not produce updates in the clients. Furthermore, since we do not know which DOM elements changed, we cannot reliably send partial updates, but only full serialised web pages. Both issues might be solved by implementing JavaScript MutationObserver, which is not available in the current version of the headless browser, but will be in the next release.
- Instead of completely removing the unneeded elements from components that only show parts of a web page, we are simply hiding them as proposed by MultiMasher and in related work [10]. Removing these elements, in fact, could cause CSS inconsistencies in the resulting views. The extraction of elements from web pages, however, would create lighter and more adaptable components that might be used in devices with limited bandwidth such as smart phones, and at the same time it would increase general scalability and performance. Possible solutions might take into consideration techniques proposed in Bila et al. [1] and Nichols et al. [21].
- In order to avoid synchronisation problems, we removed JavaScript from the client copy of the web pages: JavaScript is instead executed in the headless browser, and the results are sent back to the client devices. This is a issue for web sites like Google Maps, which heavily rely on AJAX techniques (e.g. to load new map tails during drag and drop). The user experience could be enhanced by allowing immediate JavaScript access, but we have no suggestions regarding how this could be implemented without disrupting synchronisation.

Furthermore, because of the demonstrated flexibility of Jigsaw, user studies could be conducted to evaluate the user interface as well as the general user experience. These studies could target both non technical users and expert developers, in single-user as well as multi-user collaborative scenarios. This would be useful to assess the current interaction of the various operations in the front-end, but also to gain insights into the severity of the aforementioned problem relative to not having immediate JavaScript execution in the clients.

Moreover, this infrastructure has been developed with co-located collaborative scenarios in mind. Therefore it could be used in multi-device environments like the office of the future and the interactive living room. However, Jigsaw could be extended by implementing a variety of new functionalities:

- Geographically distributed collaboration might be enhanced by taking inspiration from tools like TogetherJS¹. Therefore, the infrastructure could be extended with audio chat, instant messaging, view of other users' cursors, and other features typical of these environments.
- New types of components might be implemented. At the moment we only support PageComponents that rely on the headless browser to display content. In future work, however, Jigsaw could allow the use of pre-packaged widgets, or even provide interactive elements like post-its that do not require an external web page. Components, furthermore, could produce snapshots of the web pages, which could then be stored on the server and accessed at a later time.
- New strategies for identifying the various elements of a web site without using element IDs, XPath, or CSS Selectors, might be investigated. These would be useful for handling web sites that evolve, which are still particularly challenging as shown in both evaluations.
- In Jigsaw the concept of users is quite basic: they are implicitly identified with the connected devices. This could be improved by adding real user support, e.g. by providing login and access control mechanisms, thus investigating both security and privacy issues.

Finally, Jigsaw could be used as prototyping tool for the rapid creation of generic cross-device mashups, which would then serve as proof of concept for designing and deploying specific applications.

¹<https://togetherjs.com>

Bibliography

- [1] N. Bila, T. Ronda, I. Mohomed, K. N. Truong, and E. de Lara. Pagetailor: reusable end-user customization for the mobile web. In *Proc. MobiSys*, 2007.
- [2] B. Cheng. Virtual browser for enabling multi-device web applications. In *Proc. MD*, 2012.
- [3] O. Chudnovskyy, C. Fischer, M. Gaedke, and S. Pietschmann. Inter-widget communication by demonstration in user interface mashups. In *Proc. ICWE*, 2013.
- [4] L. Clarkin and J. Holmes. Enterprise Mashups. *MSDN Architecture Journal*, 2007. MSDN Architecture Center.
- [5] F. Daniel, F. Casati, B. Benatallah, and M. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. In *Proc. ER*, 2009.
- [6] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, and L. Yan. Distributed orchestration of user interfaces. *Inf. Syst*, 2012. 37(6):539556.
- [7] M. de Kunder. The size of the world wide web (the internet). <http://www.worldwidewebsite.com>, Oct. 2013.
- [8] D. Dearman and J. S. Pierce. It’s on my other computer!: computing with multiple devices. In *Proc. CHI*, 2008.
- [9] G. Ghiani, F. Paternò, and C. Santoro. On-Demand Cross-Device Interface Components Migration. In *Proc. MobileHCI*, 2010.
- [10] G. Ghiani, F. Paternò, and L. D. Spano. Creating mashups by direct manipulation of existing web applications. In *Proc. IS-EUD*, 2011.
- [11] G. Ghiani, F. Paternò, and C. Santoro. Push and pull of web user interfaces in multi-device environments. In *Proc. AVI*, 2012.
- [12] S. Greenberg and M. Roseman. Groupweb: A www browser as real time groupware. In *Proc. CHI*, 1996.
- [13] M. Heinrich, F. Lehmann, T. Springer, and M. Gaedke. Exploiting single-user web applications for shared editing: a generic transformation approach. In *Proc. WWW*, 2012.
- [14] M. Husmann, M. Nebeling, and M. C. Norrie. MultiMasher: A Visual Tool for Multi-Device Mashups. In *Proc. ICWE*, 2013.

- [15] S. K. Kane, A. K. Karlson, B. Meyers, P. Johns, A. Jacobs, and G. Smith. Exploring Cross-Device Web Use on PCs and Mobile Devices. In *Proc. INTERACT*, 2009.
- [16] D. Kovachev, D. Renzel, P. Nicolaescu, and R. Klamma. Direwolf - distributing and migrating user interfaces for widget-based web applications. In *Proc. ICWE*, 2013.
- [17] D. Lowet and D. Goergen. Co-browsing dynamic web pages. In *Proc. WWW*, 2009.
- [18] J. Maras, M. Stula, and J. Carlson. Reusing web application user-interface controls. In *Proc. ICWE*, 2011.
- [19] J. Melchior, J. Vanderdonckt, and P. Roy. A Model-Based Approach for Distributed User Interfaces. In *Proc. EICS*, 2011.
- [20] M. Nebeling, M. Speicher, and M. C. Norrie. Crowdadapt: enabling crowdsourced web page adaptation for individual viewing conditions and preferences. In *Proc. EICS*, 2013.
- [21] J. Nichols, Z. Hua, and J. Barton. Highlight: A system for creating and deploying mobile web applications. In *Proc. UIST*, 2008.
- [22] F. Paternò and C. Santoro. A logical framework for multi-device user interfaces. In *Proc. EICS*, 2012.
- [23] D. S. Tan, B. Meyers, and M. Czerwinski. Wincuts: manipulating arbitrary window regions for more effective use of screen space. In *Proc. CHI*, 2004.
- [24] S. Tilkov and S. Vinoski. Node.js: Using javascript to build high-performance network programs. *Internet Computing, IEEE*, 2010.
- [25] H. Wiltse and J. Nichols. Playbyplay: collaborative web browsing for desktop and mobile devices. In *Proc. CHI*, 2009.
- [26] E. Zadok, J. M. Andersen, I. Badulescu, and J. Nieh. A comparison of thin-client computing architectures, 2000.

Appendices

A

User Manual

In this appendix we will introduce the requirements for running Jigsaw, as well as explaining the installation steps and the configuration involved.

A.1 Requirements

Jigsaw is written for the Node.js environment and it is therefore multi-platform. However, it also requires the MongoDB database, the PhantomJS headless browser, and other technologies described in Section 6.1. Furthermore, MongoDB should be running with default configuration before starting the Node.js daemon: the database is automatically created, and PhantomJS is automatically started in the background.

For the clients, no particular technologies are required: any modern browser should support Jigsaw out of the box, provided it can handle HTML5 and CSS3.

A.2 Installation

The back-end administrator should observe the following procedure, therefore install in order:

1. the latest version of the Node.js environment;
2. both MongoDB and PhantomJS applications;
3. the back-end libraries, typing the command: “npm install” inside the main Jigsaw folder (the file “package.json” contains the list of libraries that will be installed);
4. the “bower” packet manager, which is used for some client side libraries, using Node with the command “npm install -g bower”;
5. the missing front-end libraries, typing: “bower install” inside the public folder in Jigsaw (the file “bower.json” contains the list of libraries that will be installed).

A.3 Configuration

Jigsaw can be started typing the usual Node.js command “node app.js” in its main folder. However, before starting Jigsaw the following should hold true:

- the PhantomJS runnable can be found inside the PATH environment variable of the system (i.e. typing “phantomjs” in a terminal should start PhantomJS);
- the MongoDB server is running with default configuration.

Once started, Jigsaw will create a default database called “mashup”, this can be changed at the bottom of the “app.js” file - each database is considered as a different cross-device mashup.

B

Evaluation Data

The following tables, divided by category of web site, present the data collected during the technical evaluation. This data is summarised and discussed in Chapter 7.

Criteria	Arts web sites					Criteria Mean	StDev
	facebook.com	youtube.com	twitter.com	imdb.com	bbc.co.uk		
Browsing						3.73	0.31
display	3	2	5	5	5	4.00	
behaviour	3	2	5	4	5	3.80	
interaction	3	3	2	4	5	3.40	0.28
Distributing						4.81	
extraction	5	5	5	5	5	5.00	
display	5	4	5	5	5	4.80	
behaviour	4	4	4	5	5	4.40	
synchronisation	5	5	5	5	5	5.00	
Mashing						4.60	0.00
linking	5	5	3	5	5	4.60	
replaying	5	5	3	5	5	4.60	
	4.22	3.89	4.11	4.78	5.00		
						Category mean 4.40	

Table B.1: Ratings of Arts web sites

Criteria	Business web sites					Criteria Mean	StDev
	paypal.com	espn.go.com	alibaba.com	finance.yahoo.com	forbes.com		
Browsing						3.67	0.80
display	5	5	5	4	3	4.40	
behaviour	5	4	5	2	3	3.80	
interaction	1	4	3	4	2	2.80	0.55
Distributing						4.35	
extraction	5	5	5	5	5	5.00	
display	4	5	4	5	5	4.60	
behaviour	5	4	5	3	3	4.00	
synchronisation	5	4	4	3	3	3.80	
Mashing						4.40	0.28
linking	5	5	5	5	3	4.60	
replaying	5	5	5	5	1	4.20	
	4.45	4.55	4.55	4.00	3.11		
						Category mean 4.14	

Table B.2: Ratings of Business web sites

Criteria	Games web sites					Criteria Mean	StDev
	ign.com	pch.com	battle.net	steampowered.com	gamespot.com		
Browsing						4.00	0.20
display	4	1	5	5	4	3.80	
behaviour	4	4	4	4	4	4.00	
interaction	4	4	4	4	5	4.20	
Distributing						4.55	0.38
extraction	5	4	5	5	5	4.80	
display	4	4	5	5	5	4.60	
behaviour	5	1	5	5	4	4.0	
synchronisation	5	4	5	5	5	4.80	
Mashing						4.40	0.00
linking	4	3	5	5	5	4.40	
replaying	4	3	5	5	5	4.40	
	4.33	3.11	4.78	4.78	4.67		
						Category mean 4.33	

Table B.3: Ratings of Games web sites

Criteria	Health web sites					Criteria Mean	StDev
	nih.gov	webmd.com	mayoclinic.com	nlm.nih.gov	myfitnesspal.com		
Browsing						4.67	0.30
display	5	5	5	5	5	5.00	
behaviour	3	5	5	5	5	4.60	
interaction	3	5	4	5	5	4.40	
Distributing						4.75	0.19
extraction	5	5	5	5	5	5.00	
display	4	4	5	5	5	4.60	
behaviour	4	5	4	5	5	4.60	
synchronisation	4	5	5	5	5	4.80	
Mashing						4.80	0.00
linking	4	5	5	5	5	4.80	
replaying	4	5	5	5	5	4.80	
	4.00	4.89	4.78	5.00	5.00		
						Category mean 4.73	

Table B.4: Ratings of Health web sites

Criteria	Home web sites					Criteria Mean	StDev
	finance.yahoo.com	yelp.com	ehow.com	gsmarena.com	reviews.cnet.com		
Browsing						4.46	0.11
display	4	5	5	5	4	4.60	
behaviour	3	5	5	5	4	4.40	
interaction	4	5	5	5	3	4.40	
Distributing						4.50	0.38
extraction	5	5	5	5	5	5.00	
display	5	5	4	4	3	4.20	
behaviour	3	5	5	5	3	4.20	
synchronisation	3	5	5	5	5	4.60	
Mashing						5.00	0.00
linking	5	5	5	5	5	5.00	
replaying	5	5	5	5	5	5.00	
	4.11	5.00	4.89	4.89	4.11		
						Category mean 4.60	

Table B.5: Ratings of Home web sites

Criteria	News web sites					Criteria Mean	StDev
	news.yahoo.com	cnn.com	huffingtonpost.com	reddit.com	bbc.co.uk/news		
Browsing						4.53	0.23
display	4	4	4	5	5	4.40	
behaviour	5	4	5	5	5	4.80	
interaction	5	4	3	5	5	4.40	
Distributing						4.90	0.19
extraction	5	5	5	5	5	5.00	
display	5	4	4	5	5	4.60	
behaviour	5	5	5	5	5	5.00	
synchronisation	5	5	5	5	5	5.00	
Mashing						4.9	0.14
linking	5	5	5	5	5	5.00	
replaying	5	4	5	5	5	4.80	
	4.89	4.45	4.55	5.00	5.00		
						Category mean 4.78	

Table B.6: Ratings of News web sites

Criteria	Science web sites					Criteria Mean	StDev
	translate.google.com	timeanddate.com	urbandictionary.com	nlm.nih.gov	cnet.com		
Browsing						4.20	0.72
display	5	5	5	5	4	4.80	
behaviour	3	5	5	5	4	4.40	
interaction	1	3	5	5	3	3.40	
Distributing						4.70	0.38
extraction	5	5	5	5	5	5.00	
display	5	5	5	5	3	4.60	
behaviour	3	5	5	5	3	4.20	
synchronisation	5	5	5	5	5	5.00	
Mashing						4.60	0.00
linking	3	5	5	5	5	4.60	
replaying	3	5	5	5	5	4.60	
	3.65	4.78	5.00	5.00	4.11		
						Category mean 4.51	

Table B.7: Ratings of Science web sites

Criteria	Shopping web sites					Criteria Mean	StDev
	amazon.com	ebay.com	netflix.com	amazon.co.uk	walmart.com		
Browsing						4.53	0.41
display	5	5	3	5	4	4.40	
behaviour	5	5	3	5	3	4.20	
interaction	5	5	5	5	5	5.00	
Distributing						4.60	0.32
extraction	5	5	5	5	5	5.00	
display	5	5	3	5	3	4.20	
behaviour	5	5	3	5	5	4.60	
synchronisation	5	5	3	5	5	4.60	
Mashing						4.80	0.00
linking	5	5	4	5	5	4.80	
replaying	5	5	4	5	5	4.80	
	5.00	5.00	3.65	5.00	4.45		
						Category mean 4.62	

Table B.8: Ratings of Shopping web sites

Criteria	Society web sites					Criteria Mean	StDev
	amazon.com	ebay.com	netflix.com	amazon.co.uk	walmart.com		
Browsing						4.53	0.41
display	5	5	3	5	4	4.40	
behaviour	5	5	3	5	3	4.20	
interaction	5	5	5	5	5	5.00	0.32
Distributing						4.60	
extraction	5	5	5	5	5	5.00	
display	5	5	3	5	3	4.20	
behaviour	5	5	3	5	5	4.60	
synchronisation	5	5	3	5	5	4.60	
Mashing						4.80	0.00
linking	5	5	4	5	5	4.80	
replaying	5	5	4	5	5	4.80	
	5.00	5.00	3.65	5.00	4.45		
						Category mean 4.26	

Table B.9: Ratings of Society web sites

Criteria	Sport web sites					Criteria Mean	StDev
	espn.go.com	sports.yahoo.com	nfl.com	bleacherreport.com	espnricinfo.com		
Browsing						3.53	0.31
display	5	4	3	3	3	3.60	
behaviour	4	3	4	3	5	3.80	
interaction	4	2	2	3	5	3.20	0.38
Distributing						4.50	
extraction	5	5	5	5	5	5.00	
display	5	5	5	4	4	4.60	
behaviour	4	3	4	5	5	4.20	
synchronisation	4	3	4	5	5	4.20	
Mashing						4.70	0.42
linking	5	5	5	5	5	5.00	
replaying	5	5	2	5	5	4.40	
	4.56	3.89	3.78	4.22	4.67		
						Category mean 4.22	

Table B.10: Ratings of Sport web sites